(12)

Scheduling for Locality in Shared-Memory Multiprocessors

Evangelos Markatos

Technical Report 457
May 1993

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Scheduling for Locality in Shared-Memory Multiprocessors

by

Evangelos Markatos

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

Supervised by

Thomas J. LeBlanc

Department of Computer Science
College of Arts and Science

University of Rochester
Rochester, New York

1993

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1993	technical report

4. TITLE AND SUBTITLE

Scheduling for Locality in Shared-Memory Multiprocessors

5. FUNDING NUMBERS

ONR/DARPA N00014-92-J-1801

6. AUTHOR(S)

Evangelos Markatos

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester, NY 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER

TR 457.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, VA 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The last decade has produced enormous improvements in processor speed without a corresponding improvement in bus or interconnection network speeds. As a result, the relative costs of communication and computation in shared-memory multiprocessors have changed dramatically, and many parallel applications do not execute efficiently on today's multiprocessors. In this dissertation we quantify the effect of this trend in architecture on parallel program performance, explain the implications of this trend on popular parallel programming models, and propose system software to efficiently map parallel programs and programming models to modern shared-memory multiprocessors. We propose new decomposition and scheduling algorithms that significantly reduce communication overhead. Our experiments over a wide variety of shared-memory multiprocessors demonstrate that the performance benefits of our scheduling-for-locality algorithms are significant, improving performance by up to 60% for some applications. We conclude that communication overhead need not dominate performance, given an appropriate programming model, multiprogramming scheduling policy, and user-level decomposition and scheduling algorithms.

14. SUBJECT TERMS	15. NUMBER OF PAGES
shared-memory multiprocessors; architecture trends; loop scheduling; lightweight thread scheduling; multiprogramming	110
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Curriculum Vitae

Evangelos Markatos was born in *Αργοστόλι* (Argostoli), Greece on February 22 1966. After finishing High School in 1983, he entered the University of Patras from which he graduated in 1988 with a diploma in computer engineering. His senior thesis (joined with Catherine Chronaki) was on performance evaluation of schedulers for database systems.

In September 1988, Evangelos started graduate school at the Computer Science Department at the University of Rochester from which he received his M.S. degree in 1990. His research deals with several areas of computer science such as parallel operating systems, real-time systems, parallelizing compilers, synchronization, scheduling, databases systems, graph theory, and lower bounds. Although it was not easy for him to choose, Evangelos finally decided to focus his research in parallel processing and write this dissertation about *Scheduling for Locality in Shared-Memory Multiprocessors.*

Evangelos has accepted a Research Associate position with the Computer Science Institute in Crete, Greece.

Acknowledgments

First, I would like to thank Tom LeBlanc, my advisor, for his helpful guidance, inspiring discussions, and his constant contributions to the research described here. Tom is always available to talk to, and has the unique ability to take us out of the dark holes that research was always getting us into.

The two research groups that Tom and I participated in, the *Coschedulers* (Mark Crovella, Prakash Das, and Cezary Dubnicki) and the *Predictors* (Mark Crovella and Ricardo Biancini) were always a constant encouragement for creation and constructive discussion. Most of the work described in chapter 3, is the result of the cooperation with the Coschedulers.

I would also like to thank Michael Scott, Robert Fowler and Bruce Arden, the other members of my dissertation committee, for their interest and feedback on my work.

I would like to thank Catherine Chronaki for the few but precious evenings we spent studying and doing research together.

Finally, I would like to thank Evangelia Melissinou, who has always been a definition of courage and dignity for me, and Pavlos Markatos for those wonderful afternoons of youth that shaped my personality.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, NSF Research Grant No. CCR-9005633, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930).

Abstract

The last decade has produced enormous improvements in processor speed without a corresponding improvement in bus or interconnection network speeds. As a result, the relative costs of communication and computation in shared-memory multiprocessors have changed dramatically. An important consequence of this trend is that many parallel applications, whose performance depends on a delicate balance between the cost of communication and computation, do not execute efficiently on today's shared-memory multiprocessors.

In this dissertation we quantify the effect of this trend in multiprocessor architecture on parallel program performance, explain the implications of this trend on popular parallel programming models, and propose system software to efficiently map parallel programs and programming models to modern shared-memory multiprocessors. Our experiments with application programs on bus-based, cache-coherent machines like the Sequent Symmetry, and large-scale distributed-memory machines like the BBN Butterfly, confirm that applications scale much better on previous-generation machines than on current machines due to the rising cost of communication. Our experiments also suggest that shared-memory programming models, which can be implemented efficiently on the machines of yesterday, do not readily port to state-of-the-art machines. As a solution, we propose new decomposition and scheduling algorithms that significantly reduce communication overhead. Our scheduling algorithms, which apply equally well to run-time libraries and parallelizing compilers, attempt to co-locate processes and data, assigning processes to processors based on the location of the data they will access. Our experiments over a wide variety of shared-memory multiprocessors demonstrate that the performance benefits of these scheduling-for-locality algorithms are significant, improving performance by up to 60% for some applications on modern machines. We conclude that communication overhead need not dominate performance on present or future multiprocessors, given an appropriate programming model, multiprogramming scheduling policy, and user-level decomposition and scheduling algorithms.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 The Problem

Recent developments in processor technology have boosted the performance of processors more than two orders of magnitude over the last decade. The evolution of floating point co-processors, and the advent of RISC processors were the main reasons for the significant improvement in processor speed. Unfortunately, these impressive improvements in processor technology were not accompanied by a similar improvement in memory or interconnection network technology. This discrepancy has resulted in a (relative) increase of the communication cost compared to the computation cost. Generally, processor speeds have improved by more than two orders of magnitude, while memory and interconnection network speeds have improved by no more than one order of magnitude over the last decade. This fact implies that communication is about an order of magnitude more expensive (relative to computation) than it was a decade ago. Such an increase in the communication cost has fundamental implications in the design and implementation of applications for parallel processors. The performance of parallel applications depends on a delicate balance between communication and computation in each multiprocessor or on the communication to computation (c/c) ratio of the multiprocessor. An increase of an order of magnitude in the relative cost of communication implies that the importance of communication is significantly enhanced. If for example an application would suffer about 5% communication overhead in previous generation multiprocessors, the same application would now suffer about $\frac{10.5}{95+10.5} = 34\%$ communication overhead, which is generally considered rather high. Such an increase in communication overhead will increase bus and memory contention, which in turn will result in even higher communication overhead. Most programmers might be willing to tolerate a 5% overhead, especially if it enables the use of a simple high-level programming model. However, when the overhead approaches 35% or more, the performance degradation may become too high to tolerate. Even when programmers are willing to tolerate such overheads, the multiprocessor is heavily underutilized and it may not justify its cost.

While architects were making communication more expensive, parallel software designers were proposing programming models that make heavy use of the communication medium (e.g. shared memory). Lightweight process libraries [5, 11, 71] encourage users to use large numbers of threads, without attempting to reduce the communication cost

associated with those threads. These programming models were successful in previous generation multiprocessors that employed slow CISC processors, and relatively fast (compared to the processor used) interconnection networks. The communication overhead in those programming environments was no more than 5%, which is rather modest. Applications that used to scale almost linearly in previous generation shared-memory multiprocessors are not able to use more than 2-3 processors in modern multiprocessors. For example, figure 2.7 shows a parallel rarefied hypersonic flow simulator, a member of the SPLASH suite of applications. This code scales very well on an Encore Multimax multiprocessor (a rather slow previous generation architecture), but cannot use more than 2 processors on a recent bus-based cache-coherent multiprocessor (a RISC based architecture). Figure 2.3 plots the speedup of Gaussian elimination, an application from numerical analysis, on several shared-memory multiprocessors. We see that the application has almost linear speedup on older multiprocessors (like the BBN Butterfly I, the Symmetry, and the Multimax), while it has very poor speedup on modern multiprocessors (like the KSR-1, the Butterfly TC2000 and the SGI Iris).

The reason applications do not scale as well as they used to is that most programming models for shared-memory multiprocessors ignore communication as a significant source of overhead. Up to a few years ago, shared-memory multiprocessors used such slow processors that even if applications made heavy use of the communication medium, the effect of communication in the total completion time of the program was almost negligible. The evolution of fast processors however, changed this picture drastically. Communication in current shared-memory multiprocessors is so visible, it becomes the dominant overhead factor in most parallel applications.

In this dissertation we quantify the extent of the communication overhead incurred by parallel applications and contrast it to the overhead the *same* applications incurred in previous-generation multiprocessors. Our study includes bus-based cache-coherent multiprocessors and large-scale NUMA (both cache-coherent and non-cache-coherent) multiprocessors. We explain the implications of this overhead for the parallel programming models used, and which of those models are obsolete. We propose novel system software support for popular programming models on modern shared-memory multiprocessors. Our system software is in the form of decomposition and scheduling algorithms for operating systems, thread libraries, and compilers. We test our proposals on several shared-memory multiprocessors, including the Butterfly family of NUMA multiprocessors, the KSR-1 large-scale cache-coherent multiprocessor, and some small-scale bus-based cache-coherent multiprocessors like the Sequent Symmetry, the Encore Multimax, and the SGI Iris.

1.2 The Solution

The thesis that drives this dissertation is:

> Communication is becoming a dominant source of overhead in shared-memory multiprocessors. Novel decomposition, scheduling and synchro-

nization algorithms are needed to reduce the ever-increasing communication overhead incurred by parallel applications in modern multiprocessors.

Because communication quickly became one of the dominant overhead factors in shared-memory multiprocessors[1] our research will focus on reducing communication in parallel applications whenever possible. In order to do that we use novel scheduling and decomposition algorithms that give particular attention to this emerging overhead dimension, without ignoring the traditional overhead dimensions of synchronization and load imbalance. We believe that scheduling and decomposition mechanisms should be implemented in two levels: the operating system kernel level and the user level, which hosts the run-time system and the compiler. There are two primary reasons why we believe a two-level approach is appropriate:

- The operating system kernel has *limited information* about the parallel application, as opposed to the compiler, or the run-time system which has lots of information available. Instead of the compiler trying to communicate this information back to the operating system, it is easier to leave some system decisions to be made by the compiler or other system software above the operating system.
- Trends in operating system research suggest that most of the resource allocation and process management software should be moved out of the operating system kernel [46, 4, 53], and implemented in user-level servers instead. The operating system only provides the basic *mechanisms* to tie the system together, while the servers provide the *policies*, which may vary from one system to another.

Kernel Scheduling

We will first examine what kind of scheduling policy is most appropriate for the operating system kernel. Although there exist many different scheduling policies, they all fall under the two broad categories of *space sharing* and *time sharing*. Under space sharing, some number of processors is dedicated to each application for a relatively long interval of time. Under time sharing, a processor may be multiprogrammed among different applications. Space sharing has good performance properties, but time sharing may be easier to implement, as it is a direct derivative of uniprocessor time sharing policies. The difference between these two approaches is most apparent when a new application arrives in the system and a reconfiguration decision has to be made. Space sharing accommodates the new application by reducing the number of processors available to existing applications, thereby reducing the physical parallelism available to those applications. Time sharing does not reduce the physical parallelism available to any application; it reduces the frequency with which physical parallelism is made available to each application. Although this difference between space sharing and time sharing may seem unimportant, it turns out to be the most influential factor in a comparison between the two families of policies, and this factor favors space sharing.

[1] We should note however, that communication has always been a dominant source of overhead in distributed memory multiprocessors and distributed systems.

Even if we settle on space sharing as the operating system scheduling policy, the compiler and run-time system are still left to solve a significant part of the scheduling and decomposition problem. We will examine this problem in the realm of thread libraries and parallelizing compilers.

Thread Library Scheduling

Thread libraries help programmers create many units of parallelism inexpensively. Although in such environments it is the user who decides the appropriate decomposition of the problem (or the number of threads used), the thread scheduling problem must be solved by the scheduler associated with the thread library. Many elaborate and scalable thread schedulers have been proposed [5, 71], but they all have one thing in common: they attempt to minimize load imbalance among processors, while keeping queue manipulation to a minimum. We believe that this emphasis on load balancing is not appropriate on modern multiprocessors. Instead, the goal of scheduling should be to minimize the total completion time of the application (which in turn maximizes throughput), which requires that we devote special attention to issues such as communication overhead and data partitioning policies. Our approach to thread scheduling, called memory-conscious scheduling, assigns threads to processors based on the location of the data that each thread will access. This approach minimizes communication overhead first, and then deals with load imbalance, and only if it actually occurs (i.e., a processor is idle during execution). Our experiments (in section 4) show improvements of up to 60% over the traditional approach which tries to minimize load imbalance. We are able to achieve such substantial performance improvements because our scheduling method reduces communication overhead along with other overhead dimensions, while previous methods ignored communication in most cases.

Compiler Loop Scheduling

Finally we consider the role of communication in compilers, and especially in the loop scheduling problem. Loop scheduling is the assignment of the iterations of a completely parallelizable loop to the processors of a multiprocessor. We show that traditional methods of loop scheduling focus on minimizing load imbalance while keeping synchronization overhead low. In our approach, we first try to reduce communication and *then* try to balance the load, while still keeping synchronization overhead down.

1.3 Contributions

The main contributions of this dissertation are:

- We identify the *increasing importance of communication overhead in modern shared-memory multiprocessors.* We show that application programmers should be concerned about the frequency of communication and the overhead of communication operations in all kinds of shared-memory multiprocessors, including bus-based

cache-coherent multiprocessors and large-scale NUMA multiprocessors. [2] Our experiments suggest that current shared-memory multiprocessors don't resemble the ideal PRAM model, where all memory locations are equidistant from all processors. Older bus-based multiprocessors were so close to the ideal PRAM model that they were called UMA (Uniform Memory Access) multiprocessors. Modern multiprocessors (both small and large scale) however, suffer from high non-local memory access costs, much more so than their UMA predecessors. This change in the importance of communication over the past few years makes it difficult to apply known research results for UMA machines in modern multiprocessors, especially if the results depend on assumptions about the frequency and cost of communication.

- We show that from the two major families of multiprogramming policies (space sharing and time sharing) *space sharing (when coupled with appropriate synchronization primitives) is preferable to time sharing* even in small-scale multiprocessors. Space sharing enables parallel applications to exploit any locality of reference they may exhibit by avoiding frequent migration of processes, and improves utilization since applications are generally able to make better use of a few fast processors, rather than many slower ones. Frequently synchronizing applications may suffer under space-sharing however, as synchronization between running and ready processes may require several context switches. We show that appropriate blocking primitives designed for multiprogrammed environments reduce this overhead significantly.

- We propose a *thread scheduling algorithm* that assigns a thread to a processor based on the location of the data it will access. We also propose a *loop scheduling algorithm* for parallelizing compilers that assigns iterations of a parallel loop to processors based on the location of data. While the thread scheduler relies on user intervention to place threads close to their data, the loop scheduler uses a heuristic assignment of iterations to processors that usually places iterations close to their data. We experiment with both algorithms on several shared-memory multiprocessors, including the Butterfly family of parallel processors, the Sequent Symmetry, and the SGI Iris shared-memory multiprocessors. We show (in chapter 5) that although our algorithms improve performance only slightly (5-10%) on the previous generation of shared-memory multiprocessors, they can improve performance by up to 60% on modern shared-memory multiprocessors.

- We show that, as currently implemented, the ***popular shared-memory programming models are not appropriate for shared-memory multiprocessors.*** Shared-memory programming models were initially designed for use on multiprocessors where communication was relatively inexpensive. The assumption of cheap communication is embedded so deeply in shared-memory programming models and their implementation that it is difficult to eradicate. For example, most operating systems for

[2] Large-scale shared-memory multiprocessors like the BBN Butterfly are often referred to as NUMA (Non Uniform Memory Access) multiprocessors because they employ a deep memory hierarchy where the cost of accessing non-local data is significantly higher than the cost of accessing local data.

bus-based shared-memory multiprocessors use a central work-queue and place all processes in the same queue. This results in processes being moved from processor to processor each time they block and resume. Most thread libraries and language run-time systems also use a central work queue as a place where all tasks reside. Message-passing programming models on the other hand, usually incur lower communication overhead and offer better performance (even on shared-memory multiprocessors), since they do not make this same assumption. So programmers of shared-memory multiprocessors should either abandon the shared-memory programming model (along with its conceptual clarity and attractive load balancing properties), or system software designers should use methods like our proposed affinity scheduling and memory-conscious scheduling to efficiently support the shared-memory programming model on shared-memory multiprocessors.

1.4 Related Work

Hardware designers have dealt with the exploitation of deep memory hierarchies using several different methods. *Caching* [7] is one method of bringing data close to processors. It has been successfully used in uniprocessor and multiprocessor systems in all levels of the memory hierarchy. Caching is usually implemented in conjunction with *prefetching*. Prefetching brings data close to a processor before it is actually needed, and has the benefit of amortizing the cost of a data transfer over several data items. Prefetching hides the latency of the interconnection network (bus), and increases its utilization.

Multiple contexts [2] have also been proposed as a way to tolerate latency in multiprocessor systems. In traditional systems, a processor does nothing else while it waits for a cache miss to be satisfied, thereby lowering processor utilization. If the processor could rapidly switch to another computation stream while waiting for a reference to be satisfied, processor utilization would increase substantially. Multiple context processors implement a fast context switch in hardware when the currently executing context blocks waiting for data. The transfer of data over the interconnection network proceeds in parallel with the execution of another context, just as computation may proceed in parallel with I/O on multiprogrammed uniprocessors.

Operating system designers have addressed the memory hierarchy by implementing software caching in multiprocessors with local and non-local memory. These operating system implementations either use daemons [39] or page faults [17, 15, 20, 43, 44, 42] to migrate and replicate pages based on reference behavior. That is, when a processor starts referencing data that is not local, the data is brought into local memory by the operating system. In effect, operating systems research in locality management deals with software caching and coherency in multiprocessors that do not have support for hardware caching and coherency. Our research on the other hand does dot assume the existence of caching or coherency at a specific level in a shared-memory multiprocessor, but is instead mainly concerned with the development of appropriate scheduling algorithms that co-locate processes and data and avoid any unnecessary inter-processor communication.

Both operating systems and hardware-based coherency schemes improve locality by bringing data close to processors as efficiently as possible. In this dissertation we will show that significant performance improvements can also be achieved by bringing processes close to data. There have been some initial attempts at solving this problem [14, 58, 61, 77] but most of the previous research focuses primarily on the complexity of the problem, while we focus on finding robust heuristics that deliver most of the performance benefits. Software policies for scheduling, placement, and data allocation are particularly suitable to this approach.

We view our approach as complementary to hardware and operating system techniques for improving locality of reference. While previous methods (like multiple-contexts and software caching) reduce the (average) cost of each non-local memory access, our work represents an attempt to use scheduling policies to reduce *the number* of non-local memory accesses.

1.5 Outline

The next chapter presents a more detailed introduction to the problem by examining the performance of several applications on several multiprocessors. We examine the performance and scalability of applications, and evaluate the extent of the communication overhead in previous generation and modern shared-memory multiprocessors. Chapter 3 presents a comparison between time sharing and space sharing. Chapter 4 presents the memory-conscious thread scheduling method we propose. Chapter 5 presents the affinity loop scheduling policy that can be used in parallelizing compilers. Finally, chapter 6 presents a short summary of our work and our conclusions.

2 Architecture Trends vs. Software Trends

2.1 Introduction

Shared-memory multiprocessors consist of processors, memory, and an interconnection network or bus. In bus-based, cache-coherent machines like the Sequent Symmetry, there is a single global memory attached to the bus. Each processor has a local cache, which brings data from the global memory as needed; cache coherence is maintained in hardware. In large-scale shared-memory machines like the BBN Butterfly family of multiprocessors, each processor has a local memory, but may access the local memory of another processor using the interconnection network. Large-scale, cache-coherent multiprocessors like the ring-based Kendall Square KSR-1 have only cache memory, which is kept coherent in hardware.

Most shared-memory multiprocessors employ off-the-shelf microprocessors. The last decade has produced enormous improvements in microprocessor speeds due to advances in VLSI and RISC technology. These improvements in processor speed can be expected to produce a corresponding improvement in application performance. However, just as increased integer performance does not produce a corresponding improvement in operating system performance [6, 57], an increase in computational power in shared-memory machines does not guarantee a corresponding improvement in application performance. Without a corresponding improvement in bus or interconnection network speeds, it may not be possible for parallel applications to realize the full benefits of any increase in computational power. As a result, many parallel applications which depend on a delicate balance between the cost of communication and computation, do not execute efficiently on today's shared-memory multiprocessors. In some cases, two orders of magnitude improvement in processor performance produce only a factor of 5 improvement in application performance. Much of the problem stems from the fact that the tradeoffs made by system and application programmers on shared-memory machines of the recent past are no longer appropriate on current or future multiprocessors. We will first quantify the architectural trends, and then consider a few of those tradeoffs.

Machine	Int	FP	Bandwidth	Nodes	MB/sec/proc	Release
BBN Butterfly I	1	0.05	1024	256	1-4	1981
Sequent Balance 8000	2	2.7	40	12	2.2-3.3	1984
Sequent Balance 21000	2	2.7	40	30	0.9-1.3	1986
BBN Butterfly Plus	8	2.4	1024	256	1-4	1986
Encore Multimax	5	4.2	100	20	5	1987
Sequent Symmetry	10	3.67	80	30	1.8-2.7	1987
BBN TC2000	62	54.5	19456	512	2.5-10	1990
SGI Power Series	98	147	64	8	8	1990
KSR 1	123	109	32768	1088	30	1991
SGI Challenge Series	300-600	440-880	1228	36	34	1993

Table 2.1: Comparison of different shared-memory multiprocessors

2.2 Architectural Trends

Table 2.1 summarizes the performance characteristics of up to three generations of shared-memory multiprocessors produced by several vendors. The first column contains the name of each machine. Column 2 ("Int") indicates the relative performance of integer arithmetic; Column 3 ("FP") indicates the relative performance of floating point arithmetic. [1] Column 4 ("Bandwidth") lists the maximum total bandwidth (in MB/sec) of the largest possible configuration of each multiprocessor. Column 5 ("Nodes") indicates the number of processors in the maximum-size configuration supported by the architecture. Column 6 ("MB/sec/proc") lists the bandwidth available to each processor in the maximum-size configuration; a range represents the sustained and peak bandwidth for each processor. The final column ("Release") indicates the year of release for each multiprocessor.

From table 2.1 we can calculate the ratio of available bandwidth to computational power for each machine by dividing the bandwidth available to each processor (column 6 in the table) by the processing power of each processor (column 2 or 3 in the table). High values for this ratio, caused by slow processors or high bandwidth, imply that communication is inexpensive, and therefore applications should scale well on the machine. Small values for this ratio imply that the processors are too fast (or too numerous) for the available bandwidth, which will probably cause poor scalability in many parallel applications. Figure 2.1 illustrates how these ratios have varied over time within architecture families. [2]

Figure 2.1 suggests that the relative costs of communication and computation have

[1] The arithmetic performance figures were computed by executing a 64×64 uniprocessor implementation of Gaussian elimination on each multiprocessor, and normalizing the results using integer arithmetic on the Butterfly I as a baseline. The figures for the Sequent Balance 8000 and 21000, and the SGI Challenge Series were estimated based on published performance figures for these machines.

[2] The figure uses integer arithmetic as the measure of processing power. Where the bandwidth is represented by a range in table 2.1, we plotted the mid-point of that range in the figure.

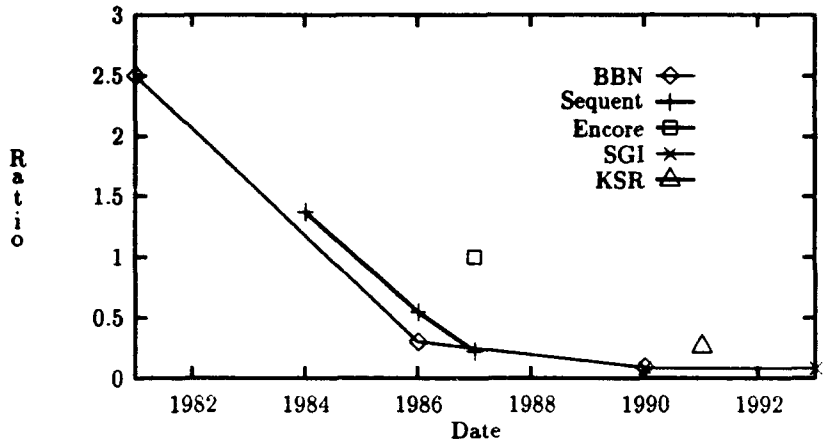

Figure 2.1: Trends in Variation in Ratio of Bandwidth to Processing Power

changed dramatically over the last ten years, primarily due to enormous improvements in integer and floating point performance. In bus-based, cache-coherent machines the ratio of (peak) bus bandwidth to processor speed has been on a downward trend from 1 : 1 in the Encore Multimax to 1 : 12 in the Silicon Graphics Power Series. Large-scale multiprocessors like the Butterfly exhibit the same trend, from a ratio of 4 : 1 in the Butterfly I to 1 : 6 in the TC2000. Although there are exceptions (such as the KSR-1), this general trend persists despite increases in communication bandwidth. The bus bandwidth of the SGI Challenge Series is 1.2 GB/sec, which is an enormous improvement over the 64 MB/sec bus in the Power Series. However, the increase in bandwidth is completely offset by a 3 to 6-fold improvement in processor speed, and a 4-fold increase in the maximum number of processors.

Some cache-coherent machines attempt to address this trend with ever larger caches. The 8 KB local caches in the Balance grew to 64 KB in the Symmetry. The Power Series has a 64 KB first-level cache and a 1 MB second-level cache. The KSR-1 has a 256 KB first-level cache and a 32 MB second-level cache (which is actually the local memory). While these larger caches can substantially improve the performance of a time-sharing job mix, the effect on parallel program performance is unclear.

Based on these trends, we cannot assume that a program that executes efficiently on the Symmetry will perform as well on a member of the Power Series. Similarly, programs that ran well on the Butterfly I and even the Plus might not run well on the TC2000. It is difficult to predict how well programs might port from a large-scale Butterfly to a small-scale member of the Power Series, or from the TC2000 to the Multimax.

2.3 Software Trends

Parallel programming environments and applications must strike a delicate balance between the costs and benefits of parallelism. The potential benefits include faster execution due to parallel hardware, and better load balancing properties due to a fine-grain decomposition of work. The costs include the overhead of *process management*, *synchronization*, and *communication*. A significant change in any of these costs affects the decision about the appropriate granularity of parallelism in an application.

In the early half of the 1980's, the overhead due to process management was a dominant factor in the decision of how best to decompose a parallel application. Kernel-implemented processes, each with a separate address space, were simply too expensive to use as a building block for parallel programming. For example, process creation in Sequent's DYNIX operating system took over 25 ms. [5], effectively precluding fine-grain parallel programming. In an attempt to support parallel programming, Mach [1] separated the kernel abstractions for address spaces and threads of control, so that a parallel program could be constructed using multiple threads within a single, shared address space. In order to avoid the overhead of entering the kernel for thread operations, several thread packages were implemented entirely in user space [12, 23, 69]. These thread libraries represent an extension of operating system functionality into user space for performance reasons. This trend continued to the point where a typical thread operation might take no longer than 10 procedure calls [5].

Very cheap threads enable fine-grain parallel programming, which in turn exposes additional sources of overhead. Synchronization, both at the application level and in the implementation of threads, became another source of concern. Synchronization often introduced significant overhead, especially when using spin-locks in cache-coherent machines. The development of efficient and scalable synchronization primitives (see [55]) dramatically reduced the cost of synchronization to the point where it need not be a significant factor in most applications.

Load imbalance was always a concern, both in small-scale, bus-based machines and large-scale machines. The two sources of load imbalance, a poor allocation of computation to threads and a poor allocation of threads to processors, were addressed using lightweight threads and a central work queue. Lightweight threads made fine-grain decomposition practical, so no one thread was responsible for a large percentage of the work to be performed. With a central work queue, no processor remains idle as long as there are threads on the queue; a thread always runs on the next available processor. Most thread packages on the Sequent and Encore machines, task bags in Linda [3], and the Uniform System library [9] on the Butterfly, are all based on these ideas. Even though central work queues have been criticized for introducing synchronization overhead (for access to the queue) [5], most systems still use this approach to load balancing.

As a result of these software trends, many parallel applications on shared-memory multiprocessors are written using a shared-memory programming model. Lightweight threads are used to represent fine-grain parallelism. Shared data is stored in a single shared address space. Efficient synchronization primitives minimize the need for non-

local references. Load imbalance is avoided by scheduling fine-grain threads using a central work queue. Every cost is under control except communication.

Communication costs refer to information transfer between processors or between a processor and memory. Cache misses, non-local memory accesses, and bulk data transfer are all communication operations in shared-memory multiprocessors. These operations have become relatively more expensive due to hardware trends, and relatively more frequent due to software trends.

2.4 Performance Impact of Hardware and Software Trends

To investigate the impact on parallel program performance we executed several parallel applications on six different shared-memory machines. The machines in our study include a 26 processor Sequent Symmetry S81, a 20 processor Encore Multimax, an 8 processor Silicon Graphics 4D/480GTX Iris, a 58 node Butterfly I, a 26 node Butterfly Plus, a 36 node TC2000, and a 64 node KSR-1. Our applications were chosen to represent various degrees of communication overhead, load imbalance, and programming effort. The issues we consider here include a comparison of fine-grain and coarse-grain decompositions, parallel versus serial communication, the effect of load imbalance, and the ease of programming in a shared-memory style.

2.4.1 Tradeoffs in Parallel Decomposition

An appropriate parallel decomposition of an application must balance several factors, including process management overhead, communication costs, the potential for load imbalance, and the ease of matching the decomposition to the algorithm. We illustrate these tradeoffs with a program to solve a system of linear equations using Gaussian elimination.

In the coarse-grain decomposition, there are P processes on P processors, and each process is allocated a subset of N/P rows in an $N \times N$ matrix. Each process performs $O(N^2/P)$ computations between successive communication operations. In the fine-grain decomposition, a process is created for each matrix element to be eliminated. Each process performs a vector addition between the pivot row and the row containing the element to be eliminated. If the rows are not in the local memory (or cache), a communication operation is required.

As seen in figure 2.2, the coarse-grain decomposition scales almost linearly on every machine, with the exception of the TC2000 and the KSR-1. The program performs well on most machines because there is little need for communication: each process executes $O(N^2/P)$ arithmetic instructions before each communication operation of size $O(N)$ (bring an entire row to local memory). Since the ratio of communication to computation is small in the application (unless P is nearly equal to N), communication overhead is not a significant factor in performance.

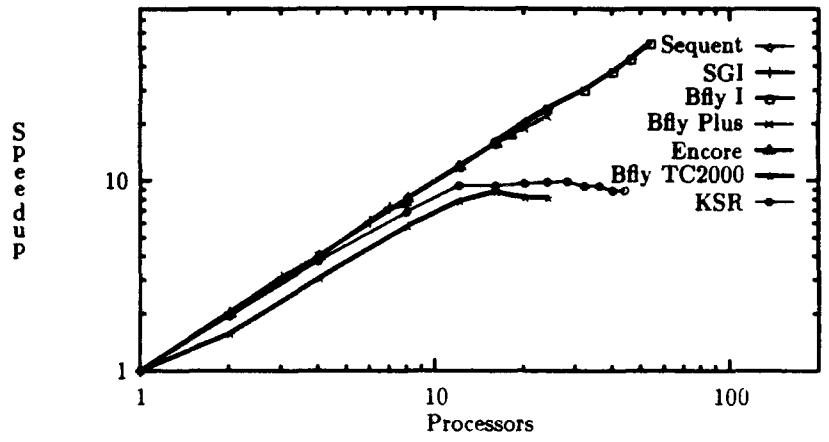

Figure 2.2: Speedup of coarse-grain Gaussian elimination

Communication is a significant problem for this application on the TC2000 and the KSR-1. The application requires that multiple processors access a pivot row serially, thereby reducing the effective bandwidth of the entire machine to the sustained bandwidth of a memory-to-memory transfer. The measured bandwidth of a memory-to-memory transfer in the TC2000 is about 10 MB/sec, and not the 38 MB/sec computed by dividing the total bandwidth in the machine by the size of the maximum configuration. On the KSR-1 the effective bandwidth between any pair of processors is around 30 MB per second [24], but the KSR-1 employs faster processors. Thus, both the TC2000 and KSR-1 perform poorly when the application requires that processors access the same data serially.

As seen in figure 2.3, the fine-grain decomposition scales extremely well on the Butterfly I, and reasonably so on the Symmetry and Multimax. This same program does not perform well on more recent machines however, reaching maximum speedup on 4 processors on the SGI Power Series machine, 6 processors on the TC2000, and 4 processors on the KSR-1. The problem is not the cost of creating and synchronizing processes (which is insignificant in this implementation), but is instead the cost of communication. Communication overhead depends on how much non-local data a process accesses compared to the time the process spends computing with the data. In the fine-grain decomposition of Gaussian elimination, each process performs a small constant number of operations on each matrix element brought in from non-local memory, whereas the coarse-grain decomposition performs a linear number of operations on each matrix element.

A comparison of the minimum achievable completion time on the BBN Butterfly

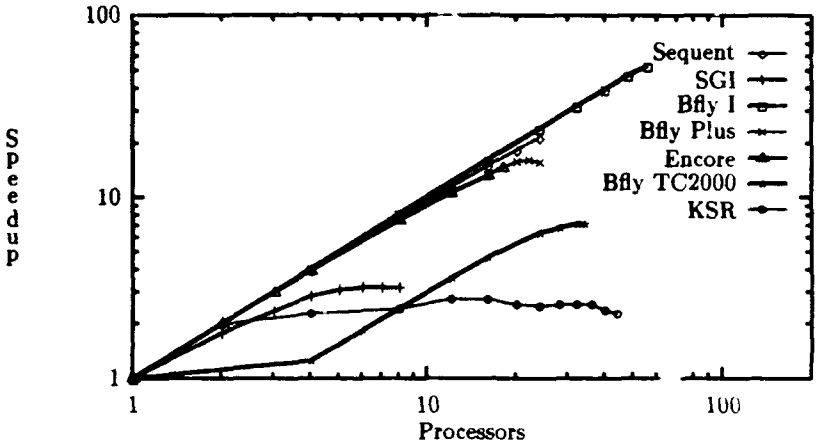

Figure 2.3: Speedup of fine-grain Gaussian elimination

(1981) and KSR-1 (1991) is particularly revealing. The coarse-grain implementation achieves a minimum completion time of 45.133 seconds on 54 processors on the Butterfly I, and 2.6 seconds on 12 processors on the KSR-1 (as seen in figure 2.4), which is a factor of 17 improvement over the Butterfly. Thus, a decade's improvement in processor speed of over two orders of magnitude improved application performance by only one order of magnitude. The situation is even worse in the case of the fine-grain implementation, where the KSR-1 improved the minimum completion time by a factor of 4.5 over the Butterfly. This and other similar applications, which performed well on earlier generations of multiprocessors, simply cannot exploit modern machines efficiently.

2.4.2 Parallel Vs. Serial Communication

Both implementations of Gaussian elimination require communication that cannot be parallelized: multiple accesses to the same pivot row must execute serially. This need for serial communication is the limiting factor on the TC2000. The Dirichlet problem, which is similar in structure to other well-known iterative computations on matrices such as successive over-relaxation (SOR), does not require serial communication, and therefore can exploit the parallel communication hardware of the Butterfly switch.

As seen in figure 2.5, the Dirichlet problem scales almost linearly on the Symmetry and Multimax, scales poorly on the TC2000, and reaches saturation on 7 processors on the SGI. On the SGI, processor efficiency is higher for Gaussian elimination than for the Dirichlet problem, since both involve comparable communication, but Gaussian elimination has more computation. There is no distinction between the inherently serial communication in Gaussian elimination and the parallel communication in the Dirichlet

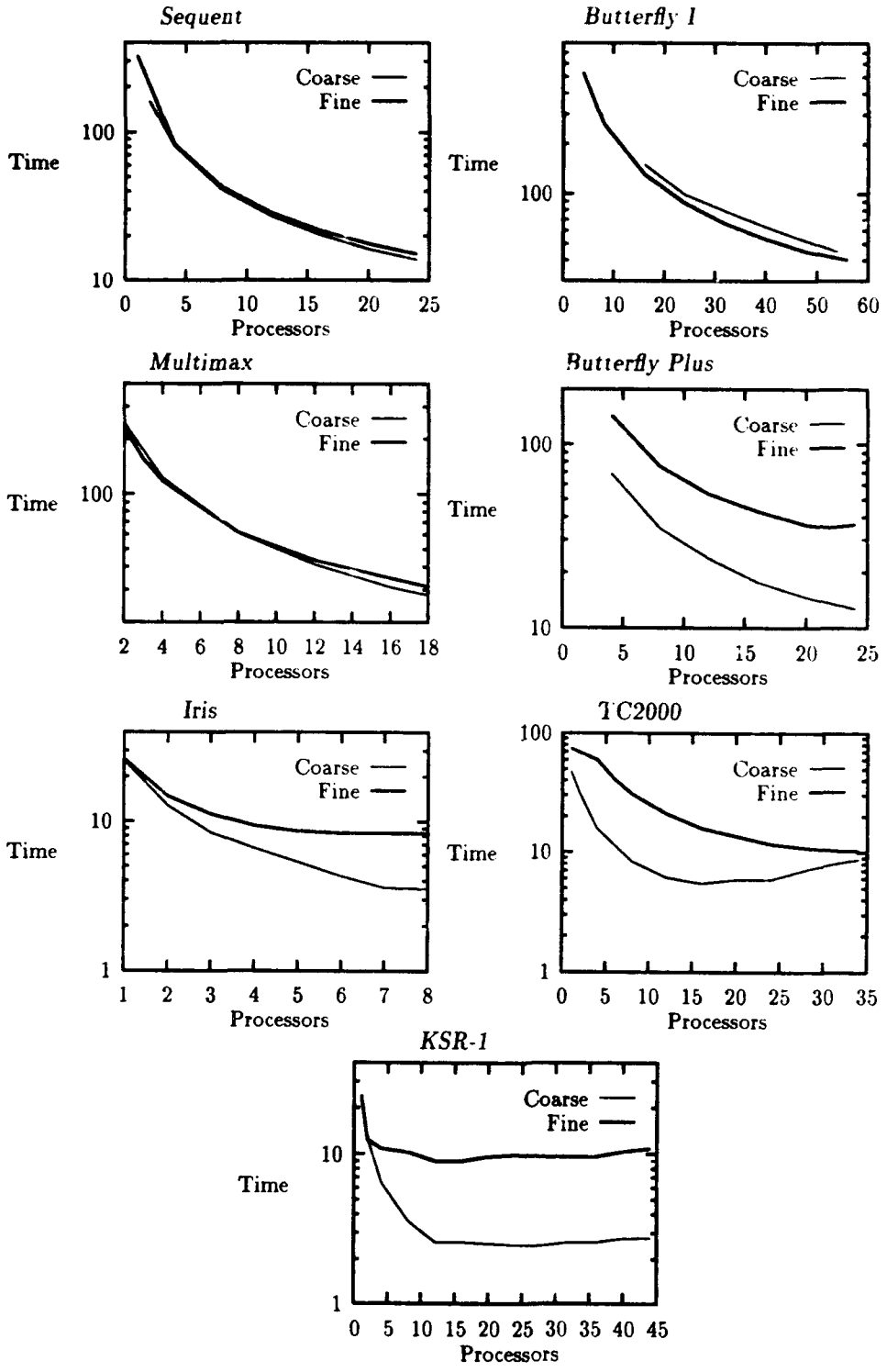

Figure 2.4: Gauss elimination on shared-memory multiprocessors

Machine	Time	Processors
Encore Multimax	29.54	20
Sequent Symmetry	12.39	24
SGI Power Series	5.34	6
BBN TC2000	2.92	36

Table 2.2: Minimum completion time (in secs) and number of processors needed to achieve this time for the Dirichlet problem.

problem on the SGI since both forms of communication employ serialized access to the hardware bus. On the TC2000, processor efficiency is slightly better for the Dirichlet problem than for Gaussian elimination; the additional computation in Gaussian elimination is offset by the serial communication, while the communication in the Dirichlet problem can be performed in parallel by the Butterfly switch in the TC2000.

Comparisons based on processor efficiency and speedup are somewhat misleading however, since completion time is the primary metric of interest. Table 2.2 presents the minimum completion time (in seconds) achieved for the Dirichlet problem and the number of processors required to achieve the minimum completion time. As seen in the table, the TC2000 is able to execute this program faster than the SGI Power Series machine, even though the SGI has more powerful processors. Unlike Gaussian elimination, the Dirichlet program, with fully parallel communication, is able to utilize 36 processors on the TC2000, which is why the TC2000 was able to produce a lower minimum completion time (albeit using 6 times as many processors).

Table 2.2 also shows that a factor of two increase in integer performance from the Multimax to the Symmetry produced a comparable improvement in application performance. A factor of 20 increase in integer performance from the Multimax to the SGI produced a factor of 5.5 increase in application performance. A factor of 12.5 improvement in integer performance from the Multimax to the TC2000 produced a factor of 10 increase in application performance. Although neither the SGI nor the TC2000 produced the full benefits of improvements in processor speed, the TC2000 was very close in this case.

2.4.3 The Effects of Load Imbalance

One reason programmers choose a fine-grain parallel decomposition is to facilitate load balancing. For those applications with the potential for large load imbalances, a fine-grain decomposition with many small pieces of work can be distributed more evenly than a coarse-grain decomposition with a few large pieces of work. In choosing a decomposition, The programmer must tradeoff load balancing properties and communication overhead. Transitive closure is an application that illustrates this tradeoff. The structure of the program is:

```
for i = 1 to N do
```


3.4.1 Time-Slicing

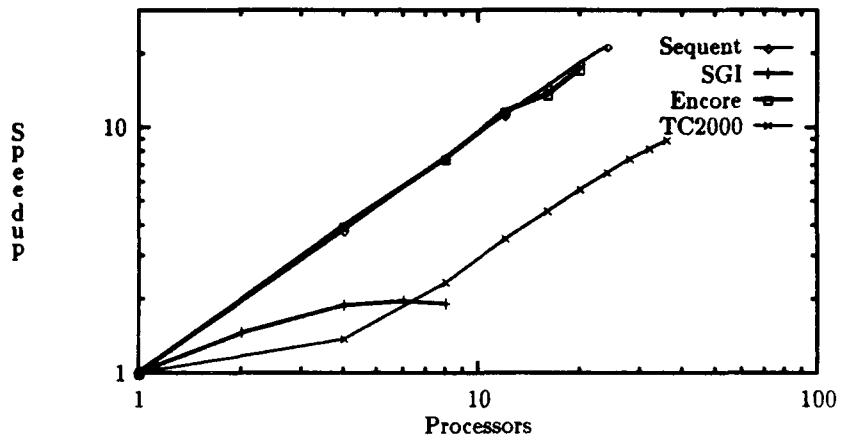

Figure 2.5: Speedup of the Dirichlet program

```
forall j = 1 to N do
      if (M(j,i)) then
            for k = 1 to N do
                  if (M(i,k)) M(j,k) := true
```

The value of the input determines whether the innermost loop is executed. Thus assigning the same number of iterations to each processor does not necessarily guarantee an even distribution of the load among processors. Assuming each iteration of the parallel loop is assigned to a different process, or some set of iterations of the parallel loop is assigned to a single processor, then the potential for load imbalance is quite high. In particular, if the input to the program is a graph of N nodes, with the first $N/4$ nodes forming a clique and the rest of the nodes having no connections, then most of the computation is contained in the first $N/4$ iterations of the parallel loop. Any static assignment of iterations to processors is likely to produce a huge load imbalance for some inputs.

The tradeoff between a coarse-grain and fine-grain decompositions of transitive closure is unclear, since it depends on the relative impact of load imbalance and communication overhead. Load imbalance (as a percentage of the total completion time) is a property of the application and the particular schedule used to assign work to processors. Communication overhead, on the other hand, is a property of the architecture. Since communication costs have risen while the effects of load imbalance remain a constant percentage of the total execution time, any performance benefits associated with a fine-grain decomposition are likely to be greater on older machines than on current machines.

Figure 2.6 shows the performance of fine-grain and coarse-grain decompositions for transitive closure. The fine-grain implementation performs slightly better than the coarse-grain implementation on the Butterfly I, and significantly better on the Symmetry. Communication is relatively cheap in both of these machines, so the benefits of load balancing dominate communication overhead. The situation is reversed on the SGI, TC2000, and KSR-1, where the coarse-grain decomposition nearly always performs better. On the SGI, the effects of load imbalance are comparable to the communication overhead; on the TC2000 and KSR-1, communication costs dominate beyond 6-8 processors.

This application illustrates how an increase in the cost of communication relative to computation has important ramifications for parallel applications. When communication was relatively cheap, as was the case throughout much of the 1980's, scheduling techniques that balanced the load while increasing communication (such as the central work queue model found in most lightweight thread packages) were beneficial. Now, even in cases where significant load imbalance is likely, communication overhead dominates, and only extreme cases of load imbalance justify incurring extra communication.

2.4.4 Ease of Programming

An important dimension that separates bus-based (Uniform Memory Access) machines from scalable (NonUniform Memory Access) machines is ease of programming. The shared-memory programming model is easy to use, and is implemented efficiently by thread libraries on most UMA machines. Earlier studies have shown that programs written for UMA machines do not execute efficiently on NUMA machines with software coherence [16]; the same holds true when porting programs from an early generation of UMA machine to the current generation of UMA machines. MP3D, a rarefied hypersonic flow simulator from the SPLASH suite of applications [64], is an example of a program that scales well on the previous generation of UMA machines, but scales very poorly on current UMA machines.

Figure 2.7 presents the completion time of MP3D on the Encore Multimax and SGI. As expected, the uniprocessor version of MP3D on the SGI is about 20 times faster than the uniprocessor version on the Multimax. However, the minimum completion time that the application achieves on the SGI is only a factor of two improvement over the minimum completion time on the Multimax. Although the SGI processors are an order of magnitude faster than the Multimax processors, the minimum completion time of the application does not improve by more than a factor of two. Previous studies of this application [19, 64] have shown why: the miss ratio of the application greatly increases with the number of processors. Since cache misses are quite costly on the SGI (relative to the speed of the processors), it is unable to execute this program efficiently. An increase in the relative cost of communication from the Multimax to the SGI produces a dramatic reduction in the speedup of the application. Thus, even though the Multimax and SGI are both bus-based, cache-coherent multiprocessors (ie., UMAs), they must be programmed using two very different styles.

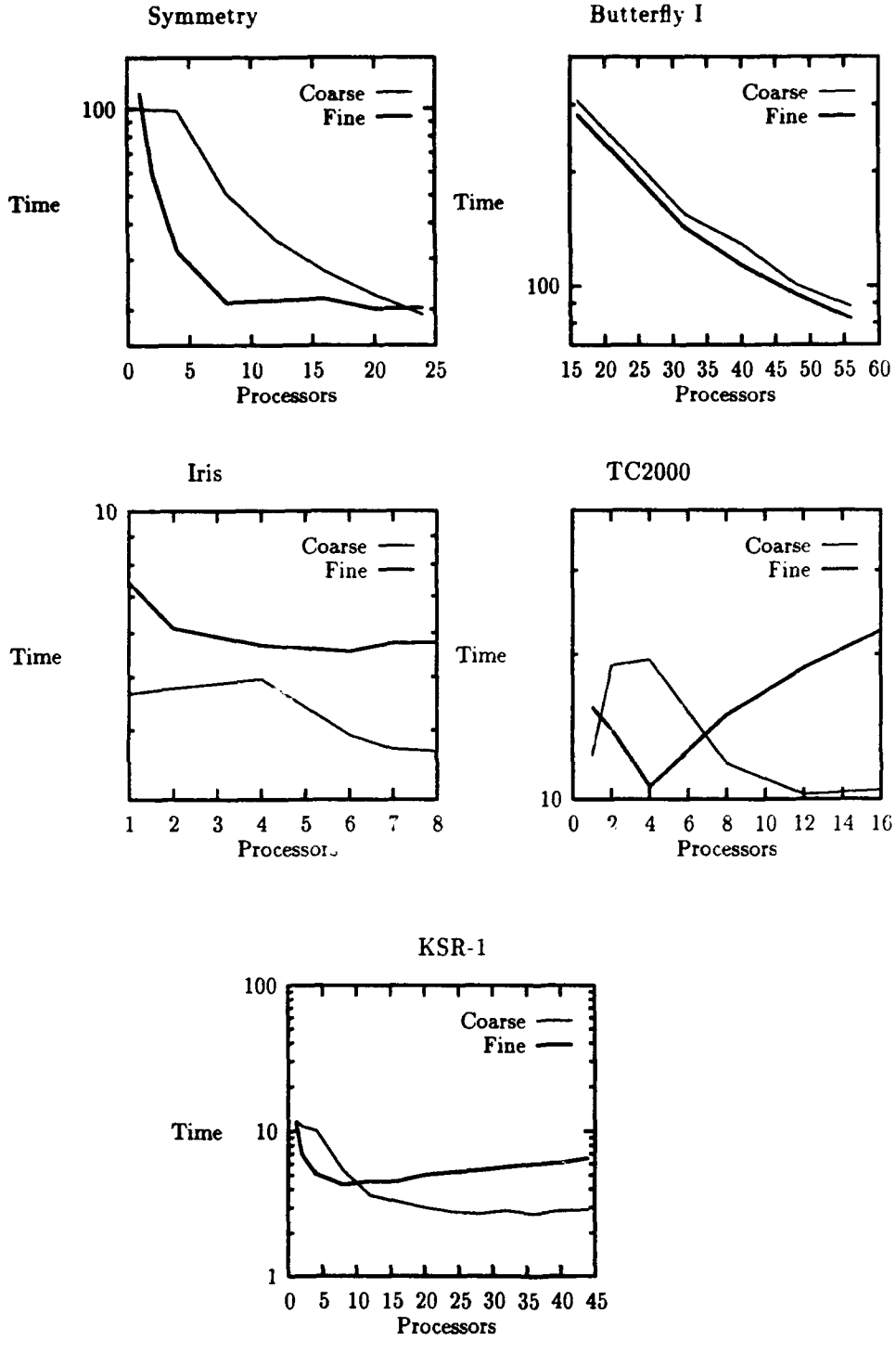

Figure 2.6: Transitive Closure

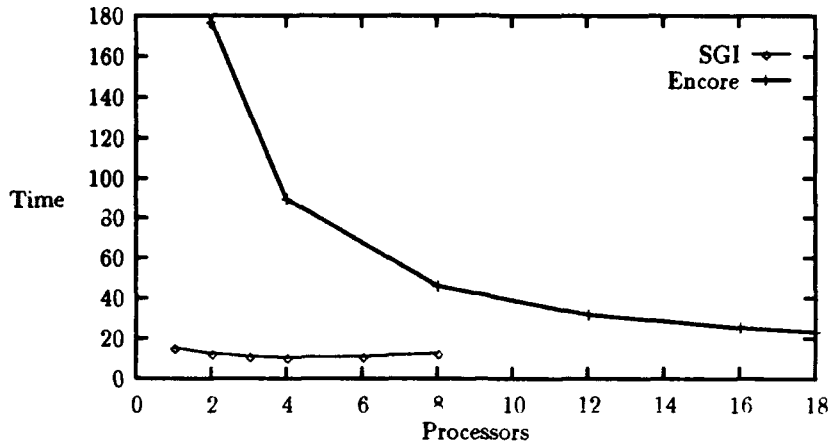

Figure 2.7: Execution time of MP3D

2.5 Implications for Software

We have argued that the overhead associated with the ever increasing cost of communication is severe and getting worse. We believe the solution lies in recognizing the extent of the problem, and modifying system and application software to reduce the need for communication.

Parallelizing Compilers

Traditional approaches to loop scheduling in parallelizing compilers attempt to distribute the computation in a parallel loop as evenly as possible, while minimizing the number of synchronization operations required [59]. The standard loop scheduling algorithms, which tradeoff synchronization costs and the effects of load imbalance, do not consider communication costs. As a result, loop scheduling algorithms that produced excellent results on the previous generation of multiprocessors suffer from excessive communication overhead on current multiprocessors.

If parallelizing compilers are to be practical for shared-memory machines, new loop scheduling algorithms that take into account the rising cost of communication are needed. Assumptions regarding the relative impact of synchronization, communication, and load imbalance need to be reconsidered. Chapter 5 describes our experiences with loop scheduling algorithms on the Iris and the KSR. Traditional loop scheduling algorithms in these multiprocessors lead to significant performance degradation. Our proposed loop scheduling algorithms, instead, lead to performance improvement by more than a factor of three in some cases.

Even when there is significant load imbalance among the iterations, our strategy improves performance by 15% or more.

Thread Packages

Thread packages leave the choice of decomposition to the programmer, but are responsible for assigning threads to processors. Our results clearly indicate that a central work queue approach, when coupled with fine-grain threads, introduces significant overhead on modern shared-memory multiprocessors. Since there is no a priori association between processors and threads when using a central work queue, the data a task will access is not likely to be in the local cache. The problem is not contention for the single work queue, but is instead the time needed to load data into the local cache each time a thread begins execution. With fine-grain threads, the percentage of execution time spent loading data into the cache is too high, up to 60% of a thread's execution time for some programs on the Iris. In several cases it is possible for the system to know where the data a thread accesses reside and it can place each thread to execute on a processor close to its data. We have used this technique (described in chapter 4) to improve the performance of fine-grain parallel programs by 30% on the Butterfly Plus and 60% improvement on the Iris.

Although a library package cannot in general know which data will be accessed by a thread, we have found that this information is readily available in BBN's Uniform System [71] programs. The necessary information can also be provided by the runtime system, as in Mercury [30], which uses object location information to schedule threads.

2.6 Conclusions

Over the last ten years, dramatic improvements in processor speeds have produced both a quantitative and qualitative change in the relationship between communication and computation in shared-memory multiprocessors. Processor speeds have improved by two orders of magnitude, while the communication bandwidth available to each processor has improved by at most one order of magnitude. If current trends continue, it will be increasingly difficult for parallel applications to utilize large-scale multiprocessors effectively.

One way to address these problems is to recognize the dominant role of communication in current systems, and to adopt techniques for reducing communication in parallel programs. Such techniques can be applied in operating system process scheduling, in user-level thread scheduling, and in loop scheduling for parallelizing compilers. We will consider each level of scheduling and its interaction with locality management in turn, in the following chapters.

3 Operating System Process Scheduling

3.1 Introduction

Multiprocessors are an expensive resource that must be shared. Sharing requires a delicate balance between fairness and resource utilization that is usually achieved through multiprogramming. In order for multiprogramming to be effective however, multiprogramming overhead must be minimized. There are several potential sources of overhead in a multiprogrammed multiprocessor environment, and each can significantly affect system performance.

Context switch overhead is introduced when processes share a processor. Even though many multiprocessor thread packages provide a user-level context switch that does not require kernel intervention, there may still be a need for several kernel processes to share a processor. The frequency of context switching through the kernel. and therefore the amount of overhead, depends on the quantum size (when processes share a processor using time-slicing) and the frequency of communication or synchronization (which may cause one process to block and another to run).

A second source of overhead is due to preemption in multiprogrammed systems that use time-slicing. If a process is preempted while inside a critical section or while computing some condition on which other processes depend, then processes may waste their quantum waiting for the preempted process to run. If processes spin while waiting, then many processor cycles are wasted by spinning. Even if processes block during synchronization, they must context switch and lose the remainder of their quantum.

A third source of overhead is the cost of cache reload, remote memory references, and migration incurred when a process is moved from one processor to another. During execution, a process builds state on a processor, either in the cache of a bus-based cache-coherent (BBCC) multiprocessor, or in the local memory of a large-scale nonuniform memory access (NUMA) multiprocessor. If the process is then assigned to another processor, it must reload the cache on a BBCC, and issue remote references or migrate the contents of memory on a NUMA. Even if a process is not moved to a different processor, other applications can corrupt the cache while it is preempted, forcing a cache reload. If the cache miss penalty is high, the associated overhead can have a serious impact on performance.

A fourth source of overhead, and one that has not received much attention, arises whenever parallel applications share processors. Every parallel program strikes a balance between the benefits of parallel execution and the overhead of parallelism *in the absence of processor-sharing*. When a multiprogramming policy causes applications to share a processor, the overhead of parallelism remains, but the effective speed of the processors appears to decrease. As a result, the balance between effective parallelism and overhead embodied in a program can be upset by the multiprogramming policy, which results in inefficient execution. In particular, an application with nonlinear speedup will prefer a small number of dedicated processors to a larger number of shared processors, even when the aggregate processor time the application receives is the same in both cases.

It is extremely difficult to find a single multiprogramming policy that can maximize processor utilization, ensure fairness, and simultaneously address all of these sources of overhead. The costs associated with a particular policy depend on the underlying architecture: the cache miss penalty, the remote access penalty, and the cost of migration. The performance implications of a policy depend on the characteristics of the applications: the number of processes per application, the amount of state associated with a process, and the frequency and type of synchronization. Due to the complexity of the problem, many of the tradeoffs inherent in multiprogramming have been examined only in the context of specific architectures and programming models, and in many cases using simulations. There has been little experimental comparison of the various solutions in the presence of applications with varying degrees of parallelism and synchronization.

In this chapter we experimentally compare the performance of three different multiprogramming schemes: time-slicing, coscheduling, and dynamic hardware partitions. We modified an existing operating system to implement the three different schemes, and then implemented several applications that vary in degree of parallelism, and the frequency and type of synchronization. Our experiments were performed on a NUMA multiprocessor without caches, but most of our results apply equally well to both BBCC architectures and multicomputers such as the Hypercube. Our results show that in most cases coscheduling is preferable to time-slicing. Our results also show that although there are cases where coscheduling is beneficial, dynamic hardware partitions do no worse, and will often do better. We conclude that under most circumstances, hardware partitioning is the best strategy for multiprogramming a multiprocessor, no matter how much parallelism applications employ or how frequently synchronization occurs.

3.2 Multiprogramming Techniques

Many different multiprogramming schemes have been proposed or implemented on multiprocessors, but most are derived from one of three basic approaches: unsynchronized time-sharing (time-slicing), synchronized time-sharing (coscheduling), and space-sharing (hardware partitions).

3.2.1 Time-Slicing

Time-slicing a multiprocessor is a straightforward adaptation of uniprocessor time-slicing, and is frequently employed in operating systems derived from uniprocessor systems. Although a single ready queue is a useful technique for balancing load across the applications in a system, multiprocessor time-sharing can suffer severe performance penalties. Since there is no guarantee that an application's processes will run at the same time, processes may be blocked while waiting for a preempted process or may be required to context switch after every synchronization operation. Several studies have shown that this effect can lead to severe performance degradation [45, 48, 49, 72]. For this reason, some systems incorporate a special mechanism to avoid preemption while in a critical section [4, 28, 53]

In addition, Squillante and Lazowska [66, 67] have shown that by ignoring the affinity that may have been created between a process and a processor, a centralized ready queue can introduce a performance penalty close to a factor of two. This penalty is attributed to cache reload costs and increased bus traffic and contention overheads. They propose a scheduling mechanism that avoids migrating processes unless significant imbalance occurs. However Vaswani and Zahorjan [74] show that affinity is generally not worth its performance benefits in current multiprocessors, and is expected to have only modest performance improvements in future multiprocessors. This seemingly apparent contradiction is based on the fact that Squillante and Lazowska were experimenting with time-sharing schedulers, while Vaswani and Zahorjan were experimenting with space-sharing schedulers (see section 3.2.3). Space-sharing schedulers migrate much less frequently than time-sharing ones, and affinity scheduling does not seem to increase their performance significantly.

3.2.2 Coscheduling

Coscheduling was originally proposed by Ousterhout [56] to address the overhead related to synchronization. With coscheduling, the processes in an application all run at the same time. There are two important advantages to coscheduling: no process is forced to wait for another that has been preempted; and processes may communicate without an intervening context switch. There are also disadvantages, however. If there are several applications in the system, the machine must cycle through each of them, during which time the caches can be expected to lose any contents related to an earlier execution [72]. Also, utilization may suffer if applications have a variable amount of parallelism, or if processes cannot be evenly assigned to time-slices of the machine. Feitelson and Rudolph [29] addressed some of the problems by implementing coscheduling in a scalable way.

Leutenegger [45] used simulation to evaluate the performance of several different policies, including coscheduling. He showed that for programs with very frequent communication, a policy that schedules the processes of an application to run at the same time performs significantly better than a policy that does not have this property. This study did not consider cache or memory affinity, or programs with a variable amount of parallelism.

3.2.3 Space-Sharing

When space-sharing (hardware partitions) is used, no two applications share a processor. A set of processors may be dedicated to an application for a relatively long fixed interval [13] or for the entire duration of the application [10]. Within its own hardware partition, each application may choose to allocate one process per processor, thereby avoiding entirely the overhead attributed to multiprogramming. However, to ensure fairness and to efficiently utilize the processors, the number of processors assigned to an application might have to change when another application arrives or departs [72], or when the degree of parallelism changes within an application [54, 81]. Unless an application can easily adjust the number of processes it employs during execution, several processes from the same application may have to share a processor, introducing context switching and other related sources of overhead.

Tucker and Gupta [72] proposed a combination of dedicated processor scheduling and a programming model that dynamically adjusts the number of processes in an application to equal the number of processors in the partition. Their experiments show that having one process per processor results in a significant performance improvement when compared to a time-slicing policy. Subsequent work by Gupta *et al.*[33] investigated the effects of different scheduling policies and synchronization primitives on an UMA multiprocessor using simulation. They showed that coscheduling and hardware partition policies are better than traditional round-robin prioritized polices due to their high cache-hit ratio and low synchronization overhead. Moreover, hardware partitions along with process control [72] typically outperform coscheduling because hardware partitions usually achieve higher processor utilization.

Unfortunately, not all applications can easily adjust the number of running processes on demand. Although the programming model used by Tucker and Gupta is widely used, their work does not characterize the effects of multiprogramming on applications that do not adhere to the model.

Zahorjan and McCann [81] simulated the performance of hardware partitions with a workload containing programs that change their parallelism frequently. They concluded that a dynamic hardware partition policy is the best choice, since such a policy can reallocate unused processors immediately. Subsequent experimental work by McCann, Vaswani, and Zahorjan [54] on a Sequent Symmetry confirms this conclusion. The same argument may not be valid for a NUMA multiprocessor however, since processor reallocation may be too expensive to perform every time an application changes the amount of parallelism it employs. In addition, applications with a fixed amount of parallelism that synchronize very frequently may prefer coscheduling over hardware partitions, since a small hardware partition may force them to incur context switch overhead on every synchronization operation.

3.3 User-Level Programming Models

There are many different parallel programming models in use today. Our goal is to explore the interactions between the programming model employed by an application

and the multiprogramming policy implemented by the operating system. Rather than attempt to cover all parallel programming languages and packages, we focus on a set of general attributes shared by many models currently in use.

We assume an application consists of a relatively large number of threads that are mapped to a relatively small number of virtual processors. Threads represent potential concurrency, while virtual processors are intended to represent true parallelism. An application has control over the number of threads used. We assume that an application is given at most one virtual processor per physical processor, but a single virtual processor may execute many threads concurrently. [1] Depending on the mapping between threads and virtual processors, threads may each have their own address space (heavyweight processes), share a single address space (lightweight processes or threads), or operate within overlapping address spaces. We assume that a context switch between threads in the same application is implemented in user space by the runtime environment of the programming model, and therefore is reasonably efficient. Context switching between virtual processors is implemented by the kernel.

The attributes of an application most directly related to multiprogramming are the ability of virtual processors to adapt in number, the number and type of threads used, and the frequency and type of synchronization.

3.3.1 Virtual Processors

Virtual processors are scheduled by the kernel. They may correspond one-to-one with physical processors, as is required in Tucker and Gupta's multiprogramming solution [72]. Alternatively, there be many more virtual processors than physical processors, as in coscheduling and time-slicing for example, wherein virtual processors from different applications share a physical processor. Some form of processor-sharing is required whenever the number of virtual processors in the system exceeds the number of physical processors.

In some applications the number of virtual processors required for execution is fixed at the time the program is written. Static parallelism is normally used to represent the functional parallelism in a program. Typically, the granularity of functional parallelism is large, and the number of virtual processors required to represent functional parallelism in a program is small. These programs are not included in our comparison.

Most applications, especially those that do not significantly vary their parallelism during execution, can be designed so that they are capable of adapting to the multiprocessor environment at the time the program begins execution. In this case the operating system can tell the application how many processors are available, and the program can create one virtual processor per physical processor. [2] No two virtual processors from the same program need share a processor, although processors might have to be shared

[1] We could multiplex several virtual processors from a single application on one physical processor, but to do so would introduce unnecessary context switching in the kernel. Because of this assumption, our measurements of the overhead of multiprogramming are conservative.

[2] Since virtual processors correspond to kernel processes, there is nontrivial overhead associated with their creation, and little reason to create more than one per physical processor.

with other applications. Once execution begins, these programs cannot easily adapt to fewer virtual processors, since to do so may require migration of threads from one virtual processor to another, and multiplexing of threads on a single virtual processor.

Some applications can adapt the number of virtual processors in use dynamically during execution, without requiring that we multiplex threads on a single virtual processor. For example, the task queue model used in Multilisp [36], Qlisp [31], BBN's Uniform System [71], Brown's thread package [23], and Chores [27], does not depend on the number of available virtual processors. Parallelism in an application is represented by tasks in the queue, which can be mapped to any number of virtual processors. The number of virtual processors can vary during execution, so long as no virtual processor is halted while executing a task.

Each class of application may exist simultaneously within a multiprogrammed multiprocessor, and will be affected to different degrees by the multiprogramming policy. Applications that can easily adapt the number of virtual processors in use may prefer dedicated processors over processor-sharing, so as to remove all multiprogramming overhead. Applications that cannot easily adapt the number of virtual processors in use, and are forced to multiplex threads on a single virtual processor, may prefer coscheduling or time-slicing over dedicated processors, so as to avoid excessive thread context switching that might arise in a small partition. The total overhead introduced by multiprogramming depends on the extent to which the affected applications are well-matched to the policy in place.

3.3.2 Threads

We assume that virtual processors are scheduled by the system and are subject to multiprogramming; threads are created and managed in user space by a thread package, and therefore are not directly under control of the operating system. Nonetheless, the number and type of thread used in an application can have an impact on multiprogramming. In particular, the costs of moving threads from one processor to another depend in part on the type of thread used in an application. In addition, the lifetime of a thread determines how cheaply we can adapt the number of virtual processors in use.

Fine-grain threads, which are typically used to represent the natural grain of parallel activity in an application, are short-lived and relatively stateless. Coarse-grain threads, on the other hand, execute longer and build up state in the cache and the local memory. As a result, a program based on fine-grain threads offers more opportunities for adaptation in a multiprogrammed environment.

Fine-grain threads come and go frequently, so any virtual processor that must give up its physical processor (either due to preemption or partition) can wait for a thread to terminate before doing so. The existence of such a clean point greatly simplifies multiprogramming with dynamic hardware partitions [72], and can be used to minimize the overhead caused by preemption during synchronization.

Coarse-grain threads do not share these characteristics. Once a coarse-grain thread creates its state on a processor, it cannot be cheaply moved to another one. If such a thread is executing when preemption occurs, there are few options to avoid the overhead

associated with preemption in the presence of synchronization. If a repartition of the physical machine is required, the cost of migration will be high.

3.3.3 Synchronization

One important source of overhead in multiprogrammed systems is due to preemption in the presence of synchronization. There are several different types of synchronization however, and each type has several implementations. The overhead introduced by preemption varies both with the type of synchronization used and the implementation. For example, preemption of programs that use spin locks could seriously affect performance, whereas preemption might not significantly affect programs that use blocking semaphores. Similarly, preemption has less impact on programs that use centralized barriers instead of tree barriers [52]; there is only one synchronization point in a centralized implementation, whereas there are two or more synchronization points in a tree barrier implementation.

The amount of overhead due to synchronization also depends greatly on the frequency of synchronization. In the worst case, an application may require a scheduling decision at every synchronization point in the program because of multiprogramming.

There are three classes of synchronization: mutual exclusion, condition synchronization, and barriers. We do not consider mutual exclusion in our experiments; it has been shown that for small critical sections that are not already a bottleneck, preemption does not impose undue overhead. The reason is that nearly all programs that use spin locks have small critical sections with low utilization, in order to scale to large numbers of processors. This implies that the probability of a process being preempted inside a critical region is rather small, and the cost associated with other processes waiting for the preempted process to complete is usually under 5% and rarely is more than 30% [80]. In addition, blocking or spinning for a short interval and then blocking if necessary, reduces the cycles lost to spinning while waiting for a preempted thread, and performs as well as pure spinning in the absence of preemption.

Our primitive for condition synchronization uses 2-phase blocking [48], where a process spins for a short while and then blocks if the condition is not satisfied, yielding the processor to another thread in the same application. This primitive has most of the performance advantages of spinning, but suffers much less in the presence of preemption. We use a tree barrier implementation that yields the processor to another thread on the same virtual processor if it exists, and spins otherwise. Although both of these primitives work well with an arbitrary number of threads, they can waste cycles spinning when applications share a processor.

3.3.4 Example Programs

Our experiments were performed using two different applications: Gaussian elimination and sorting. We chose Gaussian elimination because it has several different decompositions that allow us to measure the impact of the programming model and multipro-

gramming on the same basic application. We chose odd-even sort because it uses very frequent synchronization.

We implemented four different versions of Gaussian elimination, representing different parallelizations of row elimination. The first implementation uses stateless threads and condition (neighbor) synchronization. The main program distributes the problem matrix among the memories of the machine, creates one virtual processor per physical processor, and then creates a global queue of threads, which are assigned to virtual processors. Each thread eliminates some number of entries in the matrix. Before eliminating an entry, the thread checks to see if the condition flags associated with the pivot row and the entry are set. If so, the two rows are copied into the local memory, the computation is performed, and the result is copied back into the original matrix. When a thread terminates, a virtual processor is assigned a new thread. This program is analogous to the task queue model.

The second implementation is similar to the first, except that it uses barrier synchronization. The threads that eliminate entries in a single column of the matrix synchronize using a barrier upon completion. Then, a new set of threads is generated for the next column. The copy costs are the same in both versions.

The third implementation uses coarse-grain threads and condition synchronization. The main program creates one virtual processor per physical processor, and assigns a single thread to each virtual processor. The rows of the matrix are distributed among the threads in a round-robin fashion. Each thread eliminates all the entries in several rows. There is less synchronization than in the earlier version based on condition synchronization, since only synchronization with the pivot row is necessary. In addition, there is unlikely to be much spinning, since the elimination of the pivot row is the first computation performed in each phase of execution. Most important, there are many fewer row copy operations performed with coarse-grain threads; $O(N^2)$ instead of $O(N^3)$.

The final implementation of Gaussian elimination uses coarse-grain threads with barriers. Each thread eliminates some elements in a single column of the matrix, synchronizes with the other threads using a barrier, and then proceeds to the next column.

The sort program creates one virtual processor per physical processor, and assigns one thread to each virtual processor. The array to be sorted is divided among the virtual processors in the application. Each thread performs N/P/2 comparisons in each phase, and then synchronizes with the other threads using a barrier. The length of a phase is a few milliseconds for an array of several thousand elements on 16 nodes.

In the following section we describe the results of our experiments using these programs to compare three multiprogramming policies.

3.4 Multiprogramming Implementations

We implemented our multiprogramming experiments on a BBN Butterfly multiprocessor. We modified an existing operating system for time-slicing among applications to implement coscheduling and hardware partitions.

3.4.1 Time-Slicing

The operating system we used in our experiments on the Butterfly, Psyche [53, 62, 63], implements a straightforward extension of uniprocessor time-slicing. Users may create processes (represented by kernel processes) and bind them to physical processors. The kernel time-slices among the processes on a processor. Processes are never migrated.

Each processor has a ready queue that is sorted by process priority. Within a priority level, processes are served in a round-robin fashion. Each process gets a fair share of the processor; as in Unix, a user with many processes can get more cycles than a user with few processes.

3.4.2 Coscheduling Implementation

We implemented coscheduling using an adaptation of Ousterhout's matrix algorithm [56] and the time-slicing kernel described above. We chose this approach for simplicity, and to address two specific problems that arise when coscheduling is used in a system with priorities and blocking processes.

Kernel processes block for a variety of reasons, including while waiting for I/O or for communication with another kernel process. When a kernel process blocks, the selection mechanism used to fill unused slots in the scheduling matrix could be invoked to select another user-level process to execute the remainder of the quantum. However, if the blocked process is unblocked during the same coscheduled quantum, we would like to return the processor to that process, without overly complicating the scheduler.

We would also like to maintain the system of priorities used to implement kernel processes. Priorities are used in many operating systems to implement a fair allocation of resources and to ensure that critical operations, such as I/O, proceed immediately. We need to integrate priority scheduling and coscheduling in the same implementation.

Previous algorithms for coscheduling [29, 56] did not consider the effects of priorities or blocking kernel processes. For example, in Ousterhout's matrix algorithm, there is no efficient way to implement priorities without scanning the entire matrix on each scheduling decision. Similarly, there is no notion of whether a process is runnable or not, so the concept of yielding the processor to a runnable process is difficult to implement. For this reason, we adapted the priority-queue implementation of time-slicing to include coscheduling.

The priority range implemented in the kernel is separated into *immediate*, *coscheduled*, and *background* ranges. The highest priority processes are in the immediate range, and are assigned to kernel processes that implement I/O handlers. At any one time, the coscheduled priority range is occupied by at most one process on each processor. The background range implements a round-robin set of runnable applications.

At each quantum boundary, a single process on each processor is elevated to the coscheduled range, while the previously coscheduled process is demoted to the background range. In addition, a matrix is maintained of all jobs in the system, just as in traditional coscheduling. This matrix does not determine which processes run however,

attributed to synchronization overhead. The overhead is plotted as a function of the time between two successive barriers, expressed in multiples of the context switch time.

it only directs the promotion of processes to the coscheduled priority range. All processes occupy some place in the matrix, so no process will starve. Both priorities and process blocking are handled as in the time-slicing system. However, if a coscheduled process blocks for communication which completes before its quantum is up, that same process will receive the rest of its quantum automatically, since the coscheduled priority range is higher than the background priority range.

Coscheduling requires that process preemption be synchronized on all processors. In our implementation we use quantum of 100 *ms*. To ensure that all processors begin a new quantum simultaneously, we embed a tree barrier [78] in the clock handler of each processor. Our implementation uses a 4-way tree to combine the notification of arrivals to the barrier. Each processor is represented by a node in the tree. Each interior node waits for all its children to arrive at the barrier, and then informs its parent that it has also arrived at the barrier. When the root is so informed, it releases its children. and these in turn release their children, until all the leaves are released. Our combining tree is a 4-way tree because each processor can pack the information about its four children in one word, and with one comparison can determine if all four children have reached the barrier. The releasing tree is a binary tree.

The time required to synchronize 16 processors using this tree barrier is about 200 μs; the additional time required to make a scheduling decision using coscheduling is between 50 and 200 μs, depending on the number of applications. Without coscheduling the clock handler normally consumes about 200 μs each quantum, including the time to save state and make a scheduling decision. Our revised clock handler takes about 500 μs each quantum, or 0.5% of the quantum.

We also added two system calls to the kernel interface to support coscheduling. The first call reserves some number of slots in the coscheduling matrix for processes that are soon to be created. This call returns a handle for the application, so that when processes are created they can be added to the appropriate row of the coscheduling matrix. The second call creates a new process and places it in the coscheduling matrix in the given row.

3.4.3 Hardware Partitions

Our implementation of dynamic hardware partitions is similar to that described in [72], except that ours, being on a NUMA multiprocessor, also supports explicit migration. Our implementation requires cooperation between the operating system kernel and the library packages that implement the various programming models. The allocation of processors to applications is done in the kernel. Migration, which must occur when a partition grows or shrinks due to the departure or arrival of a new application, is implemented by the library package.

When a new application arrives or departs the system, the kernel notifies each application about changes in its hardware partition using a signal mechanism. If the partition shrinks so as to exclude a processor, the runtime library on that node may choose to either suspend the currently executing virtual processor and migrate the corresponding thread, or finish the thread and not allocate another to the virtual processor. The latter

option is used in conjunction with the task queue model; explicit migration is used in all other cases. If a thread is migrated, the underlying virtual processor is suspended, the memory object of the thread is moved to a processor in the smaller partition, and the thread is placed on the ready queue of a virtual processor in that partition.

Several system calls were added to the kernel to support hardware partitions. The *register* and *unregister* system calls are used to request and release processor partitions. A parameter to the *register* call indicates whether or not the application (or programming model) is prepared to adapt the number of processes in use if a smaller partition is provided. This information is used by the kernel when processors are allocated. The kernel tries to assign each application a fair share of processors, but no more than requested. In addition, if an application is unable to dynamically adjust the number of virtual processors in use, and the kernel cannot provide all the processors requested, then an attempt is made to balance the work across a partition. In particular, if at most x processors are available to satisfy a request for y processors, then z processors are allocated, where z is the smallest integer less than or equal to x such that: $\lceil y/z \rceil = \lceil y/x \rceil$. In this way, the ready queues on the processors in a partition remain roughly in balance, even when the application cannot adjust the number of virtual processors in use [52].

A *partition* system call returns the identity of the processors currently assigned to an application. When a thread package receives a signal from the kernel that the processor partition has changed, it uses this call to determine whether the executing processor is still in the partition. If not, it can migrate its thread to another processor in the partition.

A *migrate* system call allows for the migration of a memory object to a specified processor. The object is locked during migration, causing processors that access the object to fault, and wait until the end of the migration operation. Only thread state is ever migrated; code is replicated among the nodes in the partition, and isn't removed from a node until the associated program terminates. Also, migration can proceed in parallel on several nodes. For example, if a partition shrinks from 16 processors to 8, 8 threads can be migrated simultaneously to the remaining processors in the partition.

We currently migrate a minimum of one memory object (8K bytes) during migration. Each migration operation takes about 25 *ms* per memory object, which includes the cost of copying the memory object containing the state of the thread, unmapping the object in one address space and mapping it into another. In the worst case scenario we measured, we saw the cost of dynamically changing a system during execution from one 16-processor partition to two 8-processor partitions to be about 700 *ms*, where each process to be migrated contained 24K bytes of data.

Migration is fairly expensive in any system, and our implementation is no different. In addition, migration introduces enormous switch contention. Our implementation is sufficient for our experiments however, since (a) migration to a new partition only occurs when an application arrives or departs the system, a relatively infrequent occurrence, and (b) even relatively efficient migration is generally not helpful for short-term scheduling decisions [25].

3.5 Evaluation of Multiprogramming Policies

We ran the four different implementations of Gaussian elimination on a 16 processor BBN Butterfly in order to show how a particular programming model performs under different multiprogramming policies, and to see if there is a single multiprogramming policy that behaves best in *most* cases.

For each scheduling policy, we ran the four programs under two different scenarios: (1) under ideal conditions where only one application is in the system, and (2) under multiprogramming, with an application in the background. Our multiprogramming experiments incorporate a compute-bound application in the background that consumes any cycles it is given. Our experimental results focus on the execution time of the parallel portion of an application; the serial portion, consisting of program loading and creation of virtual processors, is not included in the timing figures.

3.5.1 Time-Slicing

Our main concern in these experiments is the overhead introduced by preemption. We first ran the two implementations of Gaussian elimination that use coarse-grain threads on a 512×512 matrix under a time-slicing policy. The running time (in seconds) of these programs on 16 processors is given in the following table:

	standalone	with background application	slowdown
coarse-grain threads, barrier synch	18.70	64.19	3.43
coarse-grain threads, condition synch	18.10	39.10	2.16

We would expect an application to take twice as long when the machine is shared with another application. In fact, a multiprogramming level of 2 introduces a slowdown of 3.43 on the program with barrier synchronization. Barrier programs are very sensitive to the effects of preemption, since the preemption of any one thread delays all threads.

The program based on condition synchronization is not adversely affected by multiprogramming. With a multiprogramming level of two, it experienced a slowdown factor of 2.16, very close to the expected. The reason that preemption does not distort this execution is that a thread does not depend on every other thread making progress during a short interval of time, as is true with barriers. Only the thread computing the next pivot row can delay other threads when preempted.

The running time (in seconds) of the fine-grain implementations is given in the following table:

	standalone	with background application	slowdown
fine-grain threads, barrier synch	38.4	97.3	2.53
fine-grain threads, condition synch	24.35	49.1	2.01

We first note that thread creation time dominates in our fine-grain thread implementation. In addition, the communication costs associated with the fine-grain implementation are much higher than in the coarse-grain implementation. The barrier program experiences a slowdown of 2.5 in this case, compared with 3.4 earlier. The reason for this apparent improvement is that both programs have the same number of barriers (and hence the same opportunities for problems with preemption), but the duration of the fine-grain thread program is greater. As a result, there are three barriers per quantum in the coarse-grain program, and fewer than 1.5 barriers per quantum in the fine-grain program. It is the frequency of barriers in the coarse-grain program that produces the difference in slowdown.

Under condition synchronization both the execution time and the slowdown of the two versions are comparable. In the case of fine-grain threads, we see once again that condition synchronization is not frequent enough in our programs for preemption to significantly affect the execution time.

None of these versions of Gaussian elimination synchronize extremely often; even the finest-grain implementation must eliminate an element between synchronization points, and that takes several milliseconds. As a result, the implementation with barriers only executes a barrier about once every 50 *ms*. Much worse cases of slowdown are possible with smaller matrices. In particular, a 256×256 matrix problem slows down by a factor of 8 in the presence of one background application.

We used our implementation of odd-even sort to measure the effect of multiprogramming on programs that synchronize very frequently. On a dedicated machine, sorting an array of 512 elements takes 286 *ms*. The same program run with a job in the background takes 103 seconds, a slowdown factor of 366! The problem is caused by a combination of barriers, frequent synchronization (every 500 μs), and preemption. If we modify the implementation of barriers to yield the processor to another application rather than spin, giving up the rest of the quantum but receiving the next quantum sooner, we see a slowdown of 540; yielding the processor ensures that almost no barrier is ever completed within a quantum.

3.5.2 Coscheduling

In order to measure the costs of coscheduling, we ran the coarse-grained Gaussian elimination programs with varying levels of processor sharing. The program used a 256×256 matrix, and was run on four processors. Processor-sharing was introduced by injecting background applications that consisted of four coscheduled threads each. The results are shown below.

Number of Processes	Running Time (secs.)	Slowdown
1	9.8	1.00
2	19.6	2.01
3	29.5	3.01
4	39.3	4.01

This table shows that as the degree of multiprogramming rises, the execution time of a single application rises linearly, despite its need for synchronization. This behavior is in contrast to the case of time-slicing.

Next we considered whether unused slots in the processor-time matrix are of any use. That is, when an application is given extra cycles for one of its threads during a time when the other threads are not coscheduled, does this improve the running time of the application? To answer this question, we ran the Gaussian elimination program with a background application that only used half the processors. This scenario creates a timeslice during which the Gaussian elimination program runs half its threads, followed by a timeslice in which all its threads run. The running time of the application in this case was 19.4 *sec*, as compared to 19.6 *sec* when sharing the machine with an application that uses all the processors during its quantum. This small difference suggests that unused slots do not contribute much to system throughput in the presence of synchronization. Ousterhout's simulations [56] show that coscheduling is typically 80% effective (measured in terms of the percent of processor time spent coscheduled); our result suggests that the effectiveness isn't improved by utilization of empty slots in the scheduling matrix.

3.5.3 Hardware Partitions

Our main concern in these experiments is the overhead introduced by multiplexing several threads on a single virtual processor and the cost of migration. To measure the overhead of multiplexing threads, we ran the Gaussian elimination program for a 512×512 matrix with 16 coarse-grain threads and barrier synchronization on 16,8,4, and 2 processors. We observed the slowdown due to having fewer processors than threads; the results are shown in figure 3.1.

We would expect the execution time of the program on 8 processors to be at least double the time on 16 processors. The additional overhead of multiplexing threads should make the time on 8 processors even more than double the time on 16 processors. Nonetheless, as shown in figure 3.1, the time required to execute the program with 16 threads on 8 processors is less than double the time used on 16 processors. These same results were observed for the program that uses condition synchronization. One reason for the better than expected performance on 8 processors is that there is significant contention for the pivot row on 16 processors, and much less on 8. Another reason is due to a slight imbalance in the computation, due to tail effects in the division of work in the matrix. In general, applications can utilize 8 processors better than 16 processors because the speedup of an application is typically sublinear.

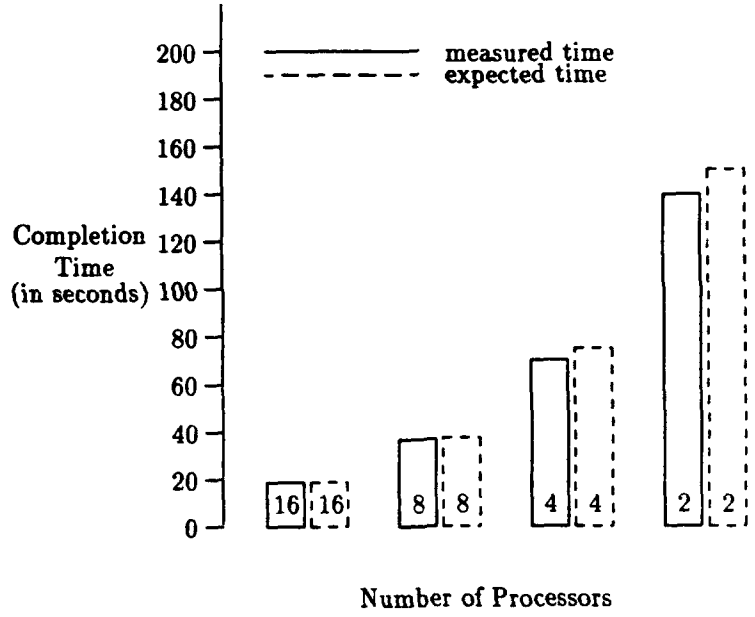

Figure 3.1: Gauss with 16 threads and barriers on hardware partitions

These experiments do not include the costs of migration. For those applications that cannot easily adapt the number of virtual processors in use, we must migrate threads when the hardware partition changes. To measure the effects of dynamic hardware partitions, we started the Gaussian elimination program on 16 processors and then immediately introduced a background application. The arrival of the second application causes the operating system to divide the machine into two 8-processor partitions. The Gaussian elimination application migrates 8 threads from the larger partition into the new smaller partition. To isolate the costs of migration, no computation was performed by the Gaussian elimination program while holding 16 processors. The completion times of the application (in seconds) are shown below.

	standalone	with background application	slowdown
coarse-grain threads, barrier synch	18.7	36.9	1.97
coarse-grain threads, condition synch	18.1	35.5	1.96

These results show that even with the one-time cost of migration, and the recurring cost of multiplexing threads on a virtual processor, a hardware partition of 8 processors takes less than twice as long as a 16 processor partition. Clearly the lack of linear speedup in the application dominates the other sources of multiprogramming overhead.

Based on this observation, we would expect the benefits of using hardware partitions to exceed the costs in most cases.

In the case of fine-grain threads, no migration or thread multiplexing is necessary. Instead, currently running threads (if any) are allowed to finish execution before removing a processor from a partition. The completion times of the application (in seconds) under dynamic hardware partitions is presented below.

	standalone	with background application	slowdown
fine-grain threads, barrier synch	38	56	1.47
fine-grain threads, condition synch	25	45	1.8

We see that, once again, the measured slowdown is much less than 2. In fact, in the case of fine-grain threads with barrier synchronization, the slowdown is only 1.47.

3.5.4 Comparison

A comparison between the three scheduling policies for each version of the Gaussian elimination program is presented in figure 3.2. Each graph contains the execution time of the program for each policy in the presence of a background application.

It is evident from these graphs that in most cases time-slicing results in slowdown much higher than the expected factor of 2. Coscheduling's slowdown is uniformly slightly higher than 2. Hardware partitions incur slowdown less than 2, and in one case substantially less than 2.

In summary, time-slicing introduces preemption, which can have enormous impact on a program, particularly programs that use barriers. Programs that don't use barriers, or synchronize infrequently are immune to the effects of time-slicing. Coscheduling has a cheap implementation and can remove the overhead due to preemption. Unfortunately, it has a built-in effectiveness of 80% or so, and performs poorly with unbalanced computations. Hardware partitions can be created fairly quickly, even when migration is required, and introduce minimal overhead due to context switching within a partition. Most important, hardware partitions allow an application to optimize its implementation for the percentage of the machine it is allocated. For this reason, hardware partitions are preferable to the other forms of multiprogramming.

3.6 The Effect of Frequent Synchronization on Space-Sharing Policies

In the previous section we showed that hardware partitions (space sharing) favor applications with sublinear speedup compared to time-sharing methods like coscheduling. It is possible however, that frequently synchronizing applications (especially programs

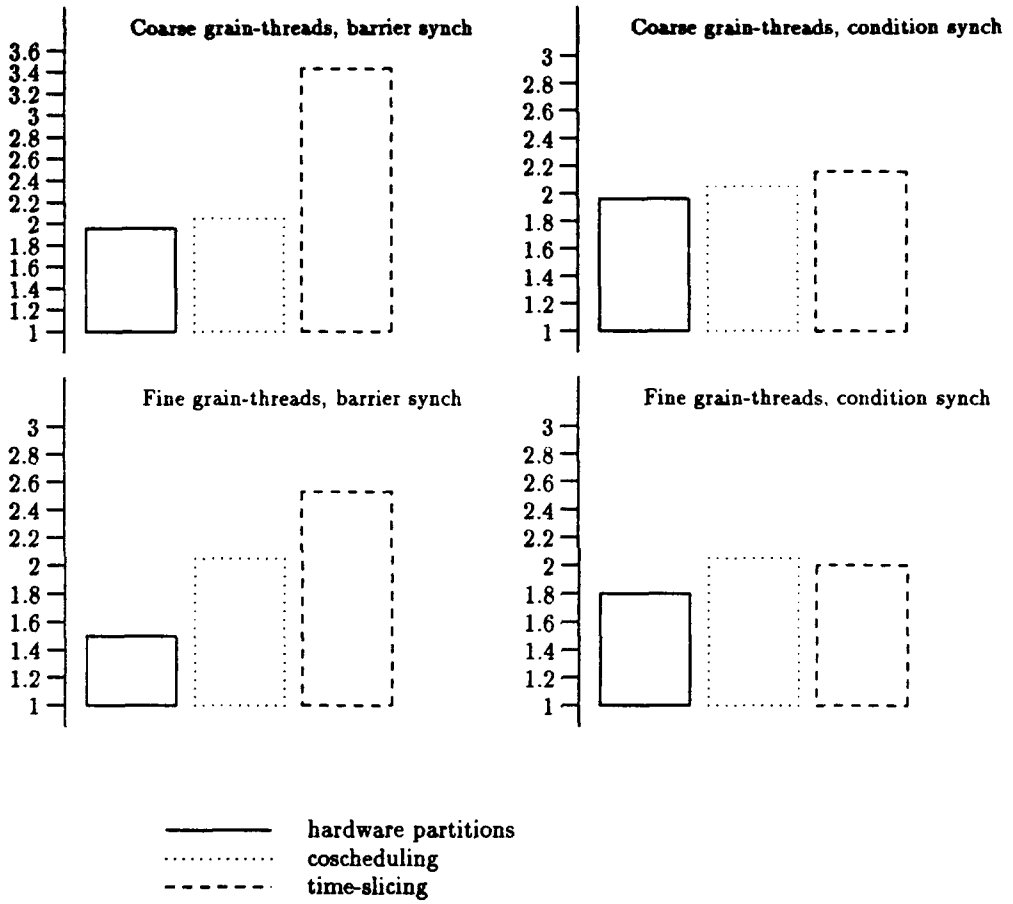

Figure 3.2: Relative slowdown introduced by a multiprogramming level of 2 for different scheduling policies and different programming models

that use barrier synchronization) suffer adverse effects under hardware partitions. In particular, as processors come and go, it is not always possible to match the number of processes with the number of processors. When processes share a processor, one process may have to context switch at a synchronization point so as to synchronize with another, preempted process. If these context switches happen frequently enough, the effect on performance could be substantial. In this section we will study the extent of this performance degradation.

3.6.1 Barrier Types

Our study is based on application programs that synchronize using barriers. There are many different implementations of barriers however, and our experience has shown that the specific barrier used can affect performance. For this reason, we consider both centralized and tree barriers, and spinning and blocking implementations of each. *Centralized barriers* use a global counter. Each process that arrives at the barrier increments the counter, and waits for the other processes to reach the barrier. When the counter is equal to the number of processes that participate in the barrier, all processes are free to proceed.

Tree barriers are representative of a large class of barriers whose completion time is logarithmic in the number of participants. This class includes tree barriers, [55, 78], tournament barriers [50], and butterfly barriers [18, 38]. Each process is represented by a node in the tree. Each interior node waits for all its children to arrive at the barrier, and then informs its parent that it has also arrived at the barrier. When the root is so informed, it releases its children, and these in turn release their children, until all the leaves are released. Our barrier combining tree is a 4-way tree. To minimize the time to wake up processes, we use a binary tree to propagate barrier completion information to the children.

Combination barriers [8] were originally suggested as a way to minimize the number of locks needed in the implementation of a tree barrier. Combination barriers incorporate both tree and centralized barriers. A centralized barrier is used for synchronization among the processes on a single processor; a tree barrier is used for synchronization across processors. If an application has as many processors as processes, then combination barriers behave exactly like tree barriers. If there are more processes than processors, then all processes on the same processor participate in a local centralized barrier. The last process on each processor to reach the local centralized barrier participates in a tree barrier with the other processors. After the tree barrier has been completed, all the waiting processes on each processor are released from the local barrier.

Centralized barriers are easy to implement and are particularly efficient in the absence of contention. Access to the counter serializes execution however, and can cause significant performance degradation in the presence of contention. Tree barriers are more complicated and less efficient for a small number of participants, but they do not serialize execution, and their performance scales logarithmically with the number of processors [55]. Combination barriers use the efficient implementation of centralized

barriers on a single node, where there is no contention, and the scalable implementation of tree barriers across processors, where there is likely to be contention.

Spinning vs. Blocking: While a process waits for others to reach a barrier, it can either spin or block. In the case of spinning barriers, a process periodically checks to see if all other processes have reached the barrier. A process will continue to spin until all processes reach the barrier or until it is preempted because of quantum expiration. In the case of blocking barriers, when a process needs to wait for an event, it yields the processor to another process of the same application running on the same processor. As an optimization, a process might yield the processor only if there is another process with which it shares the processor, spinning otherwise.

3.6.2 Barrier Performance Under Processor Deprivation

Our goal is to quantify the effect on application performance of a multiprogramming policy that allocates a barrier program fewer processors than it needs. The results depend both on the frequency of synchronization within the program and the number of processors allocated by the scheduler.

The Effects of the Frequency of Synchronization

In order to investigate the additional overhead introduced by processor deprivation (that is an application is given fewer processors than needed), we constructed an artificial program that creates M processes on top of $P(M > P)$ processors and schedules them according to a round-robin policy with local ready queues. All processes compute for the same amount of time, synchronize through a barrier, and then repeat the process. The code fragment for a process is:

```
for (int i = NUM_BARRIERS ; i>0 ; i--) {
   for (int j = MAX_DELAY ; j>0 ; j--) ;
   barrier_synchronization() ;
}
```

We vary the `MAX_DELAY` variable to change the frequency of synchronization. We vary the implementation of the `barrier_synchronization` function to reflect the different types of barriers.

We chose to have all the processes compute for the same time between successive barriers because balanced computations of this form are a worst-case scenario for a hardware partition scheduler (when compared to coscheduling). While balance in parallel programs is usually preferred, programs with imbalance in the computation can overlap computation with synchronization, reducing the effects of processor deprivation, and thereby favoring hardware partitions over coscheduling.

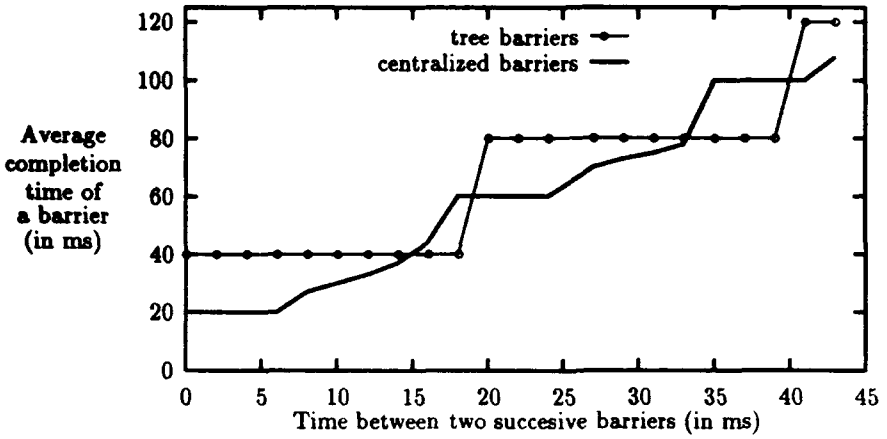

Figure 3.3: Measured execution time of tree versus centralized spinning barriers; 16 processes on 15 processors; quantum size = 20 ms.

Spinning Barriers Figure 3.3 plots the performance of an application with 16 processes on 15 processors for both tree and centralized spinning barriers. The data for figure 3.3 were produced by parallel execution of our example program on the Butterfly. The vertical axis shows the average completion time of a barrier (i.e, the average time to execute the spin loop that simulates computation in our sample program plus the time to synchronize through one barrier). We use this normalized measure of running time because it does not depend on the number of barriers a program has, but only on the frequency of synchronization and the type of barrier used. The horizontal axis shows the computation time between barriers (i.e, the time to execute the spin loop that simulates computation in our sample program). Figure 3.3 suggests that centralized barriers are better than tree barriers in nearly all cases. This result is somewhat surprising since, in the absence of processor deprivation, tree barriers perform better than centralized barriers when more than a few processors are involved [55]. This anomaly can be explained by considering the implementation of tree barriers. Tree barriers have more than one synchronization point, corresponding to the internal nodes of the tree. Information is passed up the tree as processes arrive at a barrier, and passed down again as processes are released from the barrier. Each node in the tree is effectively a barrier for its children, which must be completed before a node higher up in the tree can be notified. If a process within the tree is not running due to processor deprivation, no other process on the path up to the root can participate in the barrier. Processes close to the root may have to spin while waiting for processes lower in the tree to receive a quantum. This delay, which could be as much as a whole quantum, can be introduced between every

level in the tree.[3] Since we use a 4-way tree to pass information upwards, there are three levels in the tree used for our 16 process application, and therefore two context switches can occur between levels. Figure 3.3 shows that even if the computation time between two successive barriers is very small, the time to complete a tree barrier is still 40 *ms* (two quanta). When the time between two successive barriers is less than the quantum size, most of the quantum is spent spinning. If we slightly increase the amount of useful work done between two successive barriers, then less time is spent spinning, but the completion time of the program (in terms of quanta needed) is the same. In general, if the quantum is Q and the time between barriers is T, it takes P processes $\lceil \log_4 P \rceil \times \left\lceil \frac{T}{Q} \right\rceil$ quanta to complete a tree barrier.[4]

A centralized barrier, on the other hand, takes only 20 *ms* (one quantum) to complete when the computation time between two successive barriers is less than half the quantum. Since a centralized barrier contains only one synchronization point, the last process to reach a barrier can continue on to the next barrier within the same quantum. After the first quantum of execution, a barrier is completed every quantum.

Figure 3.3 suggests that tree barriers are better than centralized barriers when the time between two successive barriers is a little less than a multiple of the quantum size. In our simulation results [52], tree barriers are never better than centralized barriers, but the difference in performance is close to zero when the time between two successive barriers is a multiple of the quantum size. Our simulation results do not completely agree with our experimental results because the simulator does not model two important factors in the implementation of centralized barriers:

1. The time to complete a centralized barrier is assumed to be zero in the simulation. Thus, the simulation does not accurately model the linear-time complexity of centralized barriers.

2. The simulator does not model memory contention, which is much greater with centralized barriers than with tree barriers.

Both of these factors cause centralized barriers to perform worse in our experiments than in our simulation.

Blocking Barriers Blocking barriers can be used to avoid the unpredictable performance of spinning barriers in a multiprogrammed environment. With blocking barriers, a process that waits at a barrier yields the processor to another process of the same application.

[3] Adaptive tree barriers [35] do not introduce a delay at every level in the tree because all processes reside at the leaf nodes. The implementation of adaptive tree barriers in [35] would not be efficient on a NUMA machine however, since it involves excessive locking and remote spinning.

[4] In the case of 16 processes on 15 processors, the particular assignment of processes to processors is not important, since at most two context switches will be required to release at least one of the processes sharing a processor. However, if more processes or fewer processors are involved, the assignment of processes to processors can affect the number of context switches per barrier. To minimize context switching, interior nodes (or nodes close to the root of the tree) should not share a processor.

Tree Barriers: Figure 3.4 shows the results of running our test program with 16 processes on 15 processors using blocking tree barriers. Unlike spinning barriers, the performance of blocking barriers is a smooth function of the frequency of synchronization. This figure quantifies the expected superiority of blocking over spinning under processor deprivation.

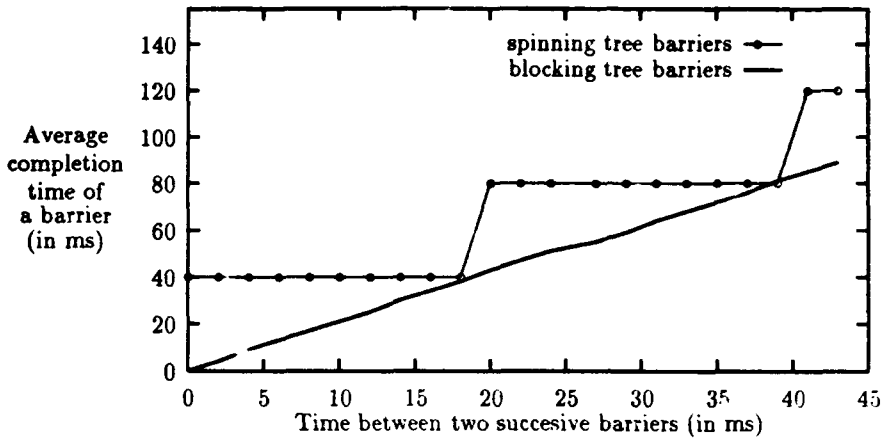

Figure 3.4: Measured execution time of spinning versus blocking tree barriers for 16 processes on 15 processors.

It is not surprising that blocking barriers perform much better than spinning barriers under processor deprivation. However, blocking barriers introduce overhead in the form of context switching. The context switch overhead may vary from as little as 10 20 μs to several hundred μs or more, depending on many factors, such as whether or not a kernel trap is required, queue manipulation overhead, the cost of saving and restoring registers, and the speed of the processors. If a single context switch is expensive, and an application synchronizes frequently, then the overhead introduced by processor deprivation could be high.

To quantify the importance of context switch overhead for blocking barriers, we ran our example program with 16 processes on 16 processors and then on 8 processors. (The results for 4, 2 and 1 processor are similar.) On 16 processors, no context switch overhead is incurred. On 8 processors, there is a context switch at every barrier. Absent context switch overhead, we would expect 8 processors to take twice as long as 16 processors to complete the program.[5] Any additional time can be attributed to context switch overhead. Figure 3.5 shows the percentage of the application's completion time

[5]Our program exhibits linear speedup, and does not adjust the number of processes to match the number of processors.

attributed to synchronization overhead. The overhead is plotted as a function of the time between two successive barriers, expressed in multiples of the context switch time.

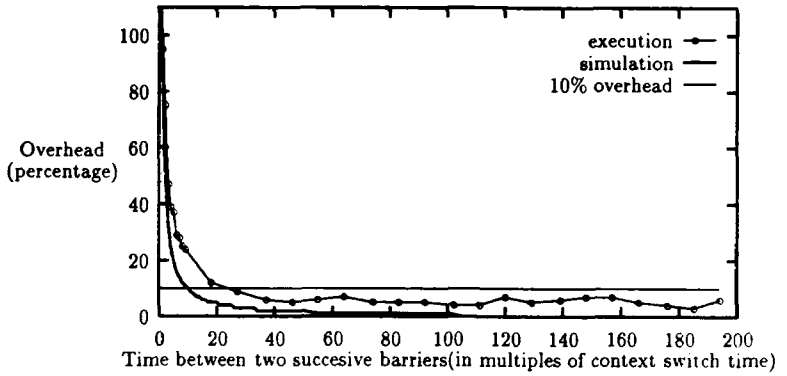

Figure 3.5: Measured overhead of blocking tree barriers for 16 processes on 8 processors.

We see from figure 3.5 that, as expected, the overhead of synchronization is inversely proportional to the computation time between barriers (i.e., the frequency of synchronization). More precisely, the overhead is proportional to the number of barriers a program has, the number of levels in the barrier tree, and the context switch cost. As shown in figure 3.5, when the time between two successive barriers is more than 20 times the cost of a context switch, the overhead incurred by context switching within blocking barriers is less than 10% of the total execution time of the program. For a typical thread package with a context switch overhead of $100\mu s$ or less, this means that the time between two successive barriers has to be less than 2 ms for the overhead to be more than 10%.

Combination Barriers: The overhead introduced by blocking tree barriers is proportional to the number of levels in the tree since, in the worst case, a context switch may be required at each level. As we saw in figure 3.5, if synchronization is very frequent, or if the cost of a context switch is high, the overhead incurred due to context switching can be quite high, as much as 158% in extreme cases. Centralized barriers induce fewer context switches, but suffer from overhead introduced by contention. Combination barriers offer a compromise solution, without the contention problems of centralized barriers, or the additional context switching of tree barriers.

- *Model 3:* The shared data a thread will access is evenly divided among three, randomly selected, memory modules.

To measure the overhead of context switching with combination barriers, we again ran our example program with 16 processes on 16 processors and then on 8 processors, this time using combination barriers. As in figure 3.5, we plotted the percentage difference in the completion time of the application on 8 processors and 16 processors. The results are shown in figure 3.6.

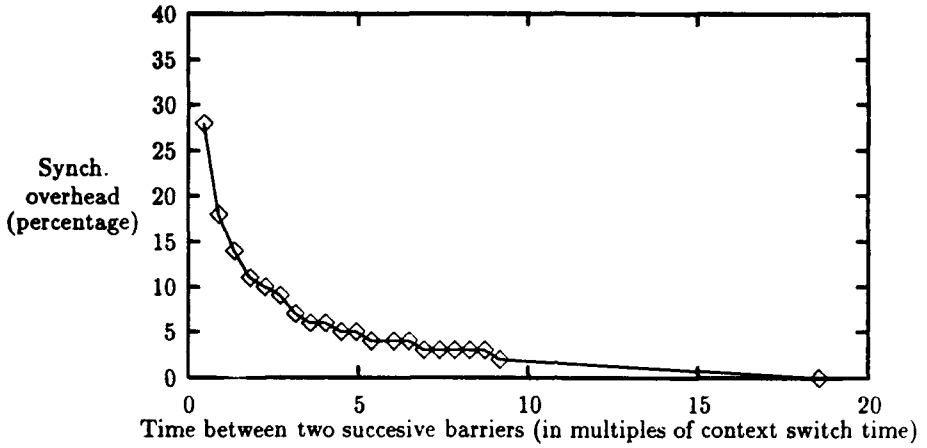

Figure 3.6: Measured overhead of blocking combination barriers for 16 processes on 8 processors.

The results are similar to those in figure 3.5, but the scale is quite different. Since combination barriers have many fewer context switches than tree barriers (in the worst case), the synchronization overhead of combination barriers is much smaller than the synchronization overhead for tree barriers. In fact, the overhead of combination barriers drops very quickly towards zero as the frequency of synchronization decreases. Figure 3.6 shows that if the time between two successive barriers is as little as 4 times the cost of a context switch, then the synchronization overhead is still less than 10%. If the time between two successive barriers is 20 times the cost of a context switch, then the synchronization overhead is practically zero. In contrast, the overhead of tree barriers is 10% when the time between barriers is 20 times the cost of a context switch, and 39% when the time between barriers is only 4 times the cost of a context switch.

Figures 3.5 and 3.6 allow us to draw some conclusions about the benefits of coscheduling, particularly in comparison to hardware partitions. In figure 3.6 we see that if the time between two successive barriers is less than the time to perform a context switch, then the overhead due to synchronization can be as high as 30%. In such cases coscheduling may be preferable to hardware partitions. If we use combination barriers, and the time between two successive barriers is more than 3 times the cost of a context switch, then the synchronization overhead is rather small. Therefore, we see that the synchro-

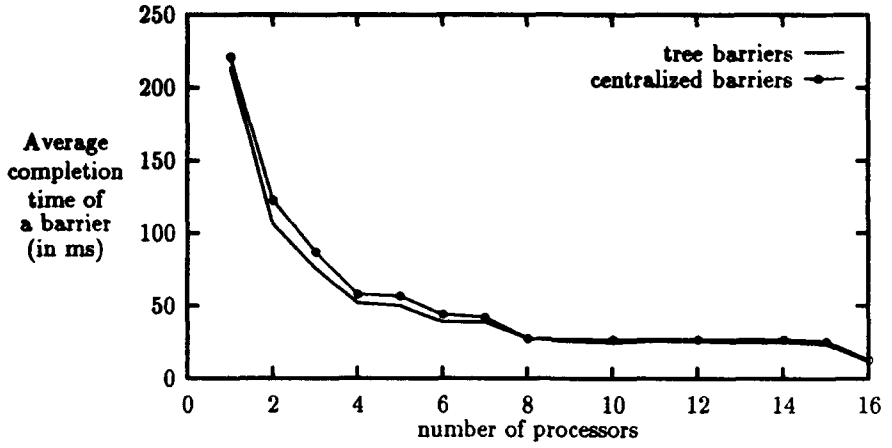

Figure 3.7: Effect of the number of processors on completion time (16 processes).

nization overhead for medium- and coarse-grain parallel programs, which represent the majority of current parallel applications, is negligible. For example, a 512×512 Gaussian elimination program (which has 512 barriers) on 16 processors synchronizes every 45 *ms* on a BBN Butterfly Plus, which is more than 450 times the thread context switch overhead. Applications such as Dijkstra's shortest path algorithm and odd-even sort synchronize every 5-10 *ms* on a BBN Butterfly for reasonable input sizes. Thus, it seems that although there are cases where coscheduling could have an advantage over hardware partitions (namely, very fine-grain synchronization, linear speedup applications), those cases are quite rare. For most applications, hardware partitions do not impose noticeable overhead, provided that appropriate blocking barriers (such as combination barriers) are used.

The Effect of the Number of Processors

We now consider what happens as we vary the number of processors allocated to an application. Since we do not allow migration, we do not expect the completion time of an application to be a smooth function of the number of processors allocated to it. For example, the completion time of an application with 16 processes running on 15 processors is about the same as the completion time of the same application running on 8 processors. Figure 3.7 shows the measured completion time (per barrier) as a function of the number of processors given to the application, for both blocking centralized barriers and blocking tree barriers.

Intuition suggests that the completion time of the application does not vary as we vary the number of processors allocated to the application between two successive

divisors of the number of processes.[6] This intuition is not entirely correct however. In the absence of migration, the completion time of an application changes only when the maximum number of processes on any processor changes. Figure 3.7 shows there is a change in completion time for both types of barriers when the number of processors given to an application changes from 16 to 15, from 8 to 7, from 6 to 5, from 4 to 3, from 3 to 2, and from 2 to 1. This result can be phrased as follows:

> If P processors are to be given to an application that has M processes that are never migrated, then the completion time of the application would be the same if it was given P_M processors where P_M is the smallest integer less than or equal to P such that $\lceil M/P_M \rceil = \lceil M/P \rceil$.

This result suggests a need for close cooperation between the kernel and user-level process management software in order to minimize the completion time of an application.

3.7 Conclusions

There are many potential sources of overhead associated with multiprogramming, and the amount of overhead from any single source depends on the structure of applications. Given an efficient implementation of context switching in user space, and relatively infrequent context switching by the kernel, the overhead due to context switching is not a serious consideration. Preemption during synchronization is of considerable concern, and has serious consequences for applications that are time-sliced. This overhead can be avoided using coscheduling or hardware partitions.

There has not been much experimental evidence showing the relative importance of overhead due to processor-sharing, in particular with respect to coscheduling. Our results show that processor-sharing in the context of coscheduling performs significantly worse than dedicated hardware partitions wherein no processor-sharing occurs. In general, there are several reasons why this will be true: (1) coscheduling results in cache corruption, whereas hardware partitions do not; (2) there are fewer remote references and less contention when fewer processors are used; (3) there is less imbalance in the computation when the total amount of work is divided among fewer processors. These factors are significant enough to more than compensate for the costs of (infrequent) migration and the additional overhead of blocking synchronization (rather than busy-waiting) required within a hardware partition when the number of threads exceeds the number of available processors. Based on our experiences, we believe the best choice for multiprogramming on a large-scale machine is to use hardware partitions.

In the next chapters we will investigate the kind of scheduling policies that are appropriate for scheduling *within* a partition.

[6] Zahorjan, Lazowska, and Eager [79] suggest that the number of processors given to an application that uses spinning barriers evenly divide the number of processes in the application, so as to eliminate spinning. However, perfect balance across processors only eliminates spinning when the time between two successive barriers is equal to the quantum size, and in any case, does not minimize completion time.

4 Thread Scheduling

4.1 Introduction

Threads are an increasingly popular structuring tool for parallel applications [12, 23, 69, 76]. Many applications decompose naturally into fine-grain units of computation; threads allow the implementation to reflect this natural decomposition. In addition, a program with fine-grain threads can run on any number of processors, and can easily adapt to a change in the number of processors (which might arise due to a repartition of the machine when a new application arrives) [72]. Also, fine-grain threads offer many opportunities to perform load balancing using inexpensive thread (re)placement instead of expensive (and maybe complicated) thread migration; each time a processor becomes idle, it can simply execute one of the threads waiting for a processor.

Despite these advantages, the degree of parallelism that can be effectively exploited by an application depends primarily on the overhead of using threads. Historically, this overhead has been dominated by the cost of thread creation, destruction, and context switching. However, recent work has shown that the cost of these thread management operations can be drastically reduced, so that threads need only be an order of magnitude more expensive than a procedure call [5]. Under these circumstances, threads should be cheap enough to use for fine-grain parallelism.

Unfortunately, the overhead associated with fine-grain threads is not limited to the cost of thread management. There is an additional cost to using threads that has not been explicitly recognized: the cost of bringing data into the local memory or cache where a thread executes. This cost, whether the result of explicit copy operations between local and remote memory on a distributed-memory machine, or the result of cache misses on a multiprocessor with coherent caches, can be substantial. On modern multiprocessors, which have extremely fast processors and relatively slow main memory, this overhead can dominate the execution time of a thread.

To illustrate the magnitude of this overhead, we implemented both a coarse-grain and fine-grain decomposition of Gaussian elimination on a Silicon Graphics Iris multiprocessor workstation. Using 6 processors, the coarse-grain decomposition (one thread per processor) requires 11.7 seconds to process a matrix with 400,000 elements; the fine-grain decomposition (one thread per element to be eliminated) takes 31.15 seconds on 6 processors. The factor of 3 difference in performance cannot be explained by the

cost of creating threads, since it only takes about one second to create, schedule and destroy all 200,000 threads used in the fine-grain decomposition. Because threads run to completion there is no additional context-switch overhead to be considered. Thus, the difference between the performance of the fine-grain and coarse-grain decompositions is mainly attributed to the time required by each thread to load its data into the local cache. In the fine-grain decomposition, each thread loads an entire row of the matrix into the local cache, and references each element of the row only once or twice. The time spent loading data into the cache in the coarse-grain decomposition is small, because each processor loads a portion of the matrix into its local cache, and then references only that portion.

This example illustrates a general problem with fine-grain threads: they do not execute long enough to amortize the cost of establishing their state in the local memory or cache. Even though thread operations may be very cheap, an implementation that uses fine-grain threads will typically perform much worse than an analogous coarse-grain implementation. As a result, programmers avoid using fine-grain threads, despite the many benefits of doing so.

In this chapter we propose a new thread scheduling technique, called *memory-conscious scheduling*, that reduces the overhead associated with bringing data to threads. The distinguishing feature of this technique is the priority placed on maintaining locality of reference. The basic approach is to schedule a set of threads that reference some of the same data on the same processor. By doing so we guarantee that only the first thread to run on a processor will have to bring a significant amount of data into the local memory or cache; other threads will be able to use the data left behind in the local memory or cache. If this assignment of threads to processors produces a load imbalance, we migrate threads to balance the load.

Our experiments on the Iris and a BBN Butterfly Plus multiprocessor confirm that this scheduling technique results in significant performance improvements for applications using fine-grain threads. In fact, the performance of an application using fine-grain threads and memory-conscious scheduling is often comparable to that of the corresponding coarse-grain implementation of the same application. Our simulations show that memory-conscious scheduling is beneficial even in cases where there is a significant variation in the computation time of threads, when the data a thread will access is spread over several memory modules, or when the remote-to-local memory access time ratio is small.

4.2 Performance Implications of Memory-Conscious Scheduling

In this section we examine the performance implications of memory-conscious scheduling on two different multiprocessor architectures: a scalable shared-memory machine without caches (BBN Butterfly Plus), and a bus-based, coherent-cache multiprocessor (SGI Iris). Using an implementation of memory-conscious scheduling on each machine, we

quantify the benefits of this scheduling technique, bound any potential improvements, and compare the results on the two architectures.

4.2.1 Description of Application Programs

For our experimental evaluation, we chose application programs whose communication patterns are representative of a large class of fine-grain parallel applications. These application programs are:

- *Gaussian elimination*: This well-known algorithm for solving a system of N simultaneous linear equations is representative of a large class of scientific applications that use vector operations. In the fine-grain decomposition, a thread is created for each element to be eliminated. Each thread adds a multiple of the current pivot row to the row of the element to be eliminated.
- *Merge sort*: This standard sorting algorithm is representative of a large class of *divide-and-conquer* problems, including convex hull, FFT, factorial, fibonacci, and the planar closest-neighbor problem. In the fine-grain decomposition, a thread is created for each recursive subdivision of the input. Each thread merges two sorted lists.
- *Grassfire*: This nearest-neighbor algorithm to compute the depth of objects in an image represented by a binary input matrix is representative of many other parallel algorithms for successive over-relaxation, convolution, edge detection, feature enhancement, and smoothing. During each iteration, the fine-grain decomposition creates a new thread for each row in the image.

We implemented three versions of each application: a coarse-grain decomposition, a fine-grain decomposition with a load balancing policy, and a fine-grain decomposition under memory-conscious scheduling. The coarse-grain decomposition creates as many threads as processors, and assigns a part of the data to each thread. Both fine-grain decompositions create one thread for each natural unit of parallelism in the application (e.g. one thread for each element to be computed in a matrix). Under the load balancing policy, a thread is assigned to the least loaded processor. [1] Under memory-conscious scheduling, a thread is assigned to a processor containing some of the data it will access.

4.2.2 Performance on the BBN Butterfly

To quantify the benefits of memory-conscious scheduling on the Butterfly, we measured the execution time of the fine-grain implementations of our applications with and without memory-conscious scheduling. We also measured the execution time of the fine-grain implementation under the load balancing policy exclusive of inter-processor communication, so as to place a bound on the benefits of any placement policy. The results (in seconds) of our experiments on 8 processors appear in Table 4.1.

[1] Ties are broken randomly.

Application	Load balancing	Memory-conscious scheduling	Lower Bound
Gauss elimination	101	71.5	67.3
Merge sort	0.95	0.75	0.6
Grassfire	93	67	59

Table 4.1: Execution time (in seconds) of fine-grain parallel applications.

As can be seen in Table 4.1, memory-conscious scheduling improves the performance of Gaussian elimination and Grassfire by about 28%, and merge sort by 21%, when compared against the traditional load balancing policy. Moreover, no other thread placement policy is likely to do much better, since memory-conscious scheduling is within 5-20% of the unrealizable lower bound, where communication is free. Given that communication is not free in practice, and also that parallel programs require some communication, these results suggest that memory-conscious scheduling provides nearly optimal thread placement for these fine-grain parallel programs.

4.2.3 Performance on the SGI Iris

In the previous section we showed that memory-conscious scheduling improves application performance by 20-30% on the Butterfly, a large-scale shared-memory machine. This result may not be surprising, in that the Butterfly is a NUMA multiprocessor (NonUniform Memory Access), where software-based locality management is essential to good performance. Software locality management has not received much attention in bus-based cache-coherent multiprocessors like the Iris however, since the existence of coherent caches creates an illusion of uniform memory access. In this section we quantify the benefits of using memory-conscious scheduling on the Iris, and show that in general these benefits depend not on the presence or lack of coherent caches, but rather on the cost of a remote memory access (or cache miss) relative to the speed of the processor.

To quantify the benefits of using memory-conscious scheduling on the Iris, we measured the execution time of the three different versions of each of our application programs: coarse-grain threads, fine-grain threads with load balancing, and fine-grain threads with memory-conscious scheduling. The results appear in Figures 4.1-4.3.

Figure 4.1 shows that the fine-grain decomposition of Gaussian elimination under the load balancing policy is 3 times slower than the coarse-grain decomposition on 7 processors. In addition, the fine-grain decomposition with load balancing is unable to exploit more than three processors; the extensive bus traffic generated by having every thread load a row of the matrix into the local cache during its short lifetime limits the number of processors that can be used effectively. Memory-conscious scheduling eliminates most of this bus traffic however, so much so that the performance of the fine-grain decomposition under memory-conscious scheduling is comparable to that of the coarse-grain decomposition. The only difference between the two is the cost of creating 200,000 threads, or about one second.

Figure 4.2 plots the results for Grassfire. Once again, fine-grain threads under memory-conscious scheduling are comparable to a coarse-grain decomposition. The improvement over the load balancing policy is not quite as dramatic as in the earlier example, but is still substantial. Similar results for merge sort are shown in Figure 4.3.

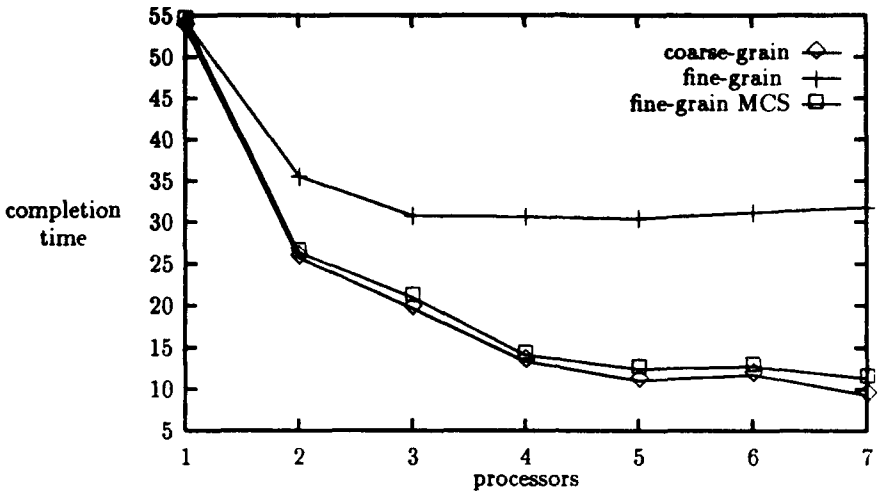

Figure 4.1: Gaussian elimination of a 640 by 640 matrix

We can draw several conclusions from these examples. First, fine-grain threads under traditional scheduling policies perform much worse than coarse-grain threads, not because of the high cost of thread management, but rather because of the cost of repeatedly loading data into the local cache. Second, memory-conscious scheduling alleviates most of this performance disparity by scheduling a thread on the processor containing the data it needs. Third, the benefits of memory-conscious scheduling depend on the application, and in some cases, on the number of processors.

The performance benefits of memory-conscious scheduling vary across our applications because each of the applications exhibits a different degree of data sharing among threads. In Gaussian elimination, each thread modifies a single row of the matrix based on the contents of the pivot row, which need only be loaded into each cache once. In merge sort, each thread processes two sorted lists produced by two other threads. In Grassfire, each thread modifies a single row of the image matrix based on the contents of two boundary rows. By scheduling a thread on the processor containing the data to be modified, memory-conscious scheduling eliminates a third of the memory traffic in Grassfire, half the memory traffic in merge sort, and nearly all the memory traffic in Gaussian elimination. The performance results in Figures 4.1-4.3 are consistent with these observations.

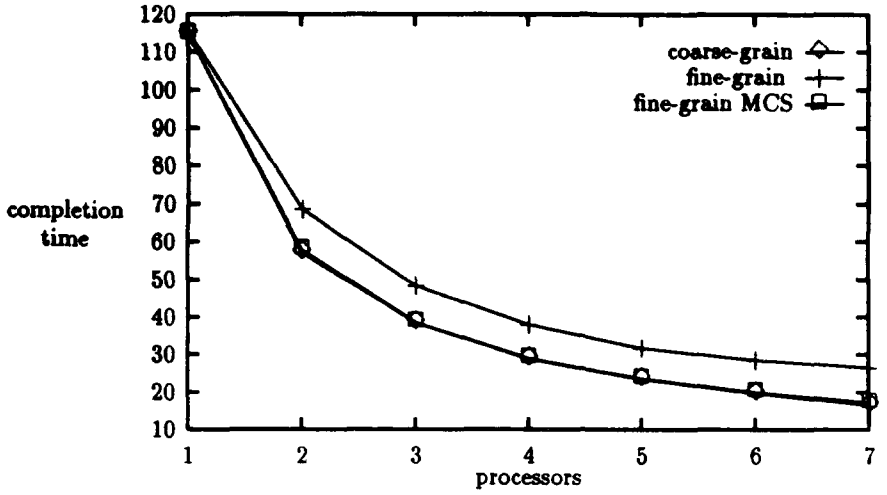

Figure 4.2: Grassfire on a 512 by 512 matrix

The Effect of Number of Processors

Figure 4.1 suggests that the relative benefits of memory-conscious scheduling depend in part on the number of processors used during execution. There are several reasons why this is true.

- With a small number of processors, there is a good chance that a random placement of threads produces the desired result of having threads run on the processors containing their data. For example, with 2 processors there is a 50% chance that a random placement policy produces the best placement for a particular thread.
- As the number of processors increases, so does bus contention, which slows down every main memory access. By reducing the need for main memory accesses, memory-conscious scheduling reduces contention, which improves the speed of any remaining accesses. This effect is particularly important on bus-based multiprocessors such as the Iris, where bus contention is often a problem (see section 2 and [60]).
- As the number of processors increases, so does the total amount of available cache (or local memory) space. Extra cache space decreases the likelihood that data will be ejected from a cache, which means that a thread's data will almost always reside in some cache.

To illustrate these points, we plotted the relative performance improvement of memory-conscious scheduling over the traditional load balancing policy for each of our

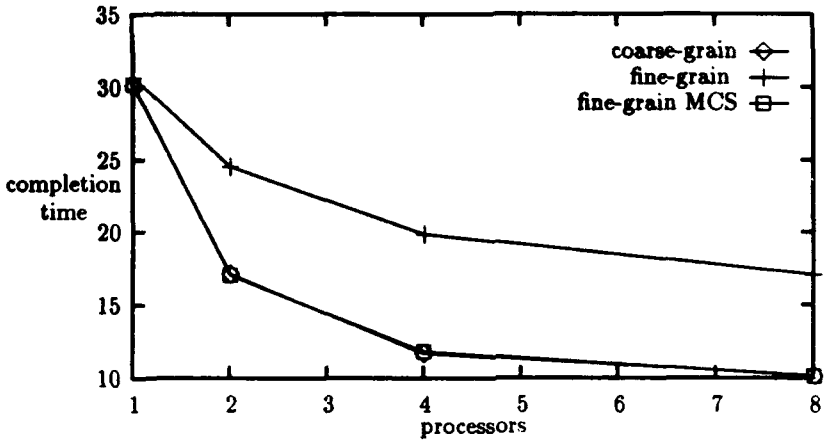

Figure 4.3: Merge sort of 2 million integers

applications as a function of the number of processors. The results appear in Figure 4.4.

As can be seen in Figure 4.4, the performance benefits of memory-conscious scheduling increase with the number of processors, although the precise improvement depends on the application. For example, the percentage improvement of Grassfire rises slowly, but steadily, with an increase in the number of processors. The improvement of merge sort rises very quickly up to 4 processors, but then remains constant. Gaussian elimination exhibits dramatic improvements up to 4 processors, and then slow, steady improvements thereafter.

Gaussian elimination exhibits a jump in improvement between 3 and 4 processors because the matrix used in our experiments doesn't fit in three caches, but does fit in four. Thus, there are no cache evictions on 4 processors; once the caches contain the entire matrix, memory-conscious scheduling reduces the need for any main memory accesses other than those caused by write-sharing.

Performance improvements are still possible even when the matrix does not fit in the local caches. As long as parts of the matrix reside in the caches long enough to be used by more than one thread, some main memory accesses are avoided. In Gaussian elimination, the portion of the matrix that needs to be stored in the caches shrinks as the matrix becomes triangular, and the computation is centered on higher numbered rows. During the latter stages of the execution, the data required by the threads will fit in the local caches, even if the original matrix did not.

The image matrix used in Grassfire fits in two caches, so we do not see much improvement as we increase the number of processors from 2 to 4. Even though the input

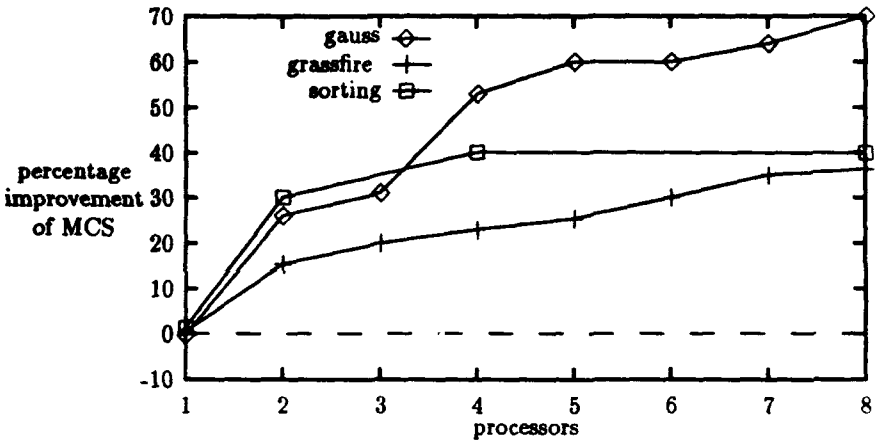

Figure 4.4: The effect of number of processors on memory-conscious scheduling.

to merge sort fits in eight caches (but not four), we do not see an improvement in the relative benefits of memory-conscious scheduling when we move from 4 to 8 processors; the form of the divide-and-conquer algorithm is such that most of the benefits of locality management come from the low levels in the tree, which do not require all the data to be resident in the caches.

In summary, we conclude that the performance benefits of memory-conscious scheduling typically increase with the number of processors. The most significant point in the performance curve usually, but not always, occurs when the number of processors is large enough so that the data of an application fits in the local caches.

4.3 Simulation Results

In order to explore the relative impact of load balancing and locality management over a wider range of parameters, including the degree of locality exhibited by an application and the cost of non-local references, we simulated the execution of a 32-node multiprocessor. In our simulation model, main memory is partitioned evenly among the processors, and is accessible to all. Thus, there is a two-level memory hierarchy. Each processor has a part of the main memory that is local to it, while the rest of the memory is remote. Each processor has its own local ready queue. The access time penalty for remote memory is a parameter of the simulation. Contention is not explicitly simulated.

We model the execution time of a thread by the number of memory references it makes. Each thread has a minimum execution time, which is the execution time of

the thread when all its references to shared data are local. We assume that 80% of the references are to private (local) data, and the remaining 20% of the references are to shared data, which may or may not be local. These percentages are similar to those measured in actual programs. The results based on these percentages can be scaled to yield the corresponding results for programs where the percentage of shared data references is larger or smaller; the performance of an application with P% private references on a multiprocessor with remote-to-local access ratio R is the same as the performance of an application with 80% private references on a multiprocessor with remote-to-local access ratio of $\frac{4RP}{1-P}$.

To introduce imbalance in the workload, we used two different models for the execution times of threads. In the first model, the execution time of a thread is drawn from a uniform distribution $[0 \cdots N]$. In the second model, the execution time of a thread is a Bernoulli random variable, where 90% of the threads make N references, and the other 10% make $100N$ references.

Threads will typically access shared data from remote memory, and so the actual execution time of a thread will usually be much larger than its minimum execution time. For each thread and processor pair, we record the percentage of references by the thread to the memory of that processor. For non-local references, this percentage is multiplied by the remote-to-local access time ratio. The time required by each processor is the sum of the execution times of the threads it runs; the time required by the application is the maximum time required by a processor.

The parameters in our simulations are the distribution of thread execution times, the assignment policy, the model of locality, and the remote-to-local access time ratio. Since the workload is determined using a random number generator, the results of each experiment are not deterministic; we ran each experiment 50 times and report the average result. All of our results were within 5% of the average.

In all our experiments we simulated an application with 1000 threads running on a multiprocessor with 32 processors. (Results of experiments with more processors and fewer threads were similar to the results reported here.) In each case, we varied the remote-to-local access time ratio, and plotted the completion time of the program as a function of this ratio.

4.3.1 Assignment Policies

To compare the effects of load balancing and locality management policies, we simulated two different thread assignment policies:

- *Load Balancing (LB):* A thread is assigned to the processor with the shortest ready queue. Once execution begins, a thread runs to completion on a single processor.
- *Memory-Conscious Scheduling (MCS):* A thread is assigned to a processor whose local memory contains most of the data the thread will access. Once started, a thread runs to completion.

If the thread assignment policy results in load imbalance, then thread reassignment is needed to keep idle processors busy. Reassignment might produce even greater load imbalance however, since a slight load imbalance among threads making local references on a processor could result in a huge load imbalance if one of those threads were assigned to another processor and forced to make non-local references. We consider the following reassignment policies:

- *Aggressive Migration (AM):* When a processor is idle, it finds the processor with the longest ready queue, removes the first thread from that queue, and executes it.
- *No Migration (NM):* Once a thread is assigned to a processor, it is never reassigned. If a processor is idle with an empty ready queue, it stays idle.
- *Beneficial Migration (BM):* When a processor is idle, it searches the ready queue of the processor with the heaviest load, as measured by the computation time of the threads in the queue. It selects a thread that will lower the execution time of the application if executed immediately on the idle processor. If no such thread is found, the processor remains idle. This policy requires complete information about the data reference patterns and computation times of threads. Because of the excessive knowledge this policy requires, it is very difficult to implement, but can serve as a basis for comparison with the other scheduling policies.

AM and NM are endpoints in the spectrum of load balancing policies. NM never reassigns a thread, even in the presence of load imbalance. AM attempts to balance the load as soon as a processor becomes idle. Although this is a reasonable approach in many cases, an aggressive policy can also cause even more load imbalance by increasing the number of non-local references made by a thread. BM is a compromise policy that reassigns a thread only if doing so improves the execution time of the application. We use this unrealizable policy, which requires more information than is generally available, as a standard to measure the effectiveness of the other policies.

4.3.2 Models of Locality

The effectiveness of locality management is partially determined by the reference behavior of threads, and in particular, the degree of sharing exhibited by the program. If a set of threads share some data, then either all those threads must execute on the same processor, or some threads will not have the data in local memory. To represent different degrees of sharing, we simulated three models of locality for shared data:

- *Model 1:* All the shared data a thread will access reside in a single, randomly selected, memory module.
- *Model 2:* Half of the shared data a thread will access resides in one, randomly selected, memory module. The other half of the shared data resides in a single, preselected memory module that is the same for all threads.

- *Model 3:* The shared data a thread will access is evenly divided among three, randomly selected, memory modules.

4.3.3 Uniform Execution Times

In our first set of experiments, all threads have uniform minimum execution times. The results of these experiments are shown in Figures 4.5 - 4.7.

In Figure 4.5, each thread accesses shared data from a single, randomly selected, memory module. Not surprisingly, MCS performs much better than LB, since the initial assignment used by MCS ensures all references will be to local memory, while the initial assignment used by LB will almost always ensure that all references to shared data will be to non-local memory.

As expected, the longest ready queue under LB has about 32 threads. When using MCS for thread assignment, the longest ready queue contains about 43 threads. The resulting load imbalance suggests the need for thread reassignment. Among the thread reassignment policies, BM and AM perform about the same, and are always better than NM. BM performs a little better than AM only when the remote-to-local access ratio is very high.

In Figure 4.6, each thread accesses shared data in two different memory modules. One memory module is randomly selected, while the other is the same for all threads. When the references are equally split among two memory modules. MCS randomly assigns the thread to one of them. Since half the shared references of each thread are directed to the same memory module, we expect *half* the threads to be assigned to that processor. As a result, the longest ready queue under MCS should contain over 500 threads, while the longest ready queue under LB still contains only 32 threads.

Even though the initial thread assignment decision made by MCS produces tremendous load imbalance, the reassignment policy is more than able to compensate for that decision. As can be seen in Figure 4.6, MCS is generally better than LB. Since there is one heavily loaded processor in this case, aggressive migration (AM) and beneficial migration (BM) perform exactly the same, since both policies immediately reassign threads to idle processors. MCS without migration (NM) is not visible in the graph due to the lack of a reassignment policy, which causes the performance of this policy to be out of scale.

In Figure 4.7, each thread accesses a third of its shared data from each of three randomly selected memory modules. In this case, at least 14% of all references must be non-local, regardless of where a thread executes. Once again, MCS is better than LB, since under MCS a third of all references to shared data will be to local memory. Also, AM and BM still perform about the same.

These simulation results suggest that processor affinity, not load balancing, should be given highest priority during thread assignment, even when an assignment based on processor affinity might produce significant load imbalance. The simulation results also suggest that a simple reassignment policy that requires little effort or knowledge of the application (AM) works as well as a more sophisticated, and less practical, policy (BM).

at most $1/P_{th}$ of the remaining work to be done.

4.3.4 Geometric Execution Times

In the following experiments, 90% of the threads have a minimum completion time of 1, and 10% of the threads have a minimum completion time of 100. We would expect this wide variation in completion times to produce a much greater load imbalance than before, and highlight the importance of load balancing policies.

Figure 4.8 compares the LB and MCS policies where the shared data accessed by a thread resides in a single, randomly selected, memory module. In this figure, the references made by a thread are either all local, or 80% local and 20% non-local. Once again, MCS with some form of thread reassignment performs better than LB. Also, for the first time, beneficial migration (BM) is noticeably better than aggressive migration (AM), up to 20% better when the remote-to-local access ratio is high. BM performs better in this case because it knows the execution time of threads and can compute the amount of load imbalance that exists before and after thread reassignment. AM assumes that thread reassignment to alleviate load imbalance will improve execution time, which may not be true if a long-running thread is forced to make non-local references as a result of reassignment.

Figure 4.9 shows the results when the shared data accessed by a thread resides in two memory modules, one of which is the same for all threads. Again MCS is better than LB as an assignment policy, while BM is about the same as AM. Since there is only one overloaded processor, both reassignment polices make the same decisions: move a thread from the overloaded processor to an idle processor. Once again, MCS without migration (NM) is off the scale due to the lack of a reassignment policy.

Figure 4.10 shows similar results for the case where the shared data a thread will access is distributed across three memory modules. MCS without thread reassignment again performs the worst, since the benefits of processor affinity are small relative to the potential load imbalance that can result from the large variation in thread execution times.

Taken together, these simulations confirm that locality management (i.e., processor affinity) should take precedence during thread assignment, but that load imbalance and locality must both be considered during thread reassignment. In particular, reassigning a thread from an overloaded processor to an idle processor is not always appropriate, especially when there is a high variation in thread execution times, or when the remote-to-local access ratio is high.

4.4 Conclusions

In this chapter we considered the problem of assigning the threads of a fine-grain parallel application to processors in a shared-memory multiprocessor. We showed that using threads to represent fine-grain parallelism can introduce excessive overhead, because each thread spends a large percentage of its lifetime bringing the data it needs into the local memory or cache. To reduce the overhead associated with fine-grain threads, we proposed a scheduling policy that places threads close to their data. Application performance improved by 20-30% when using memory-conscious scheduling

Uniform Execution Times

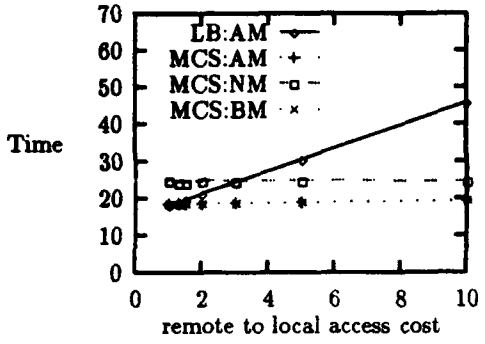

Figure 4.5: Locality Model 1.

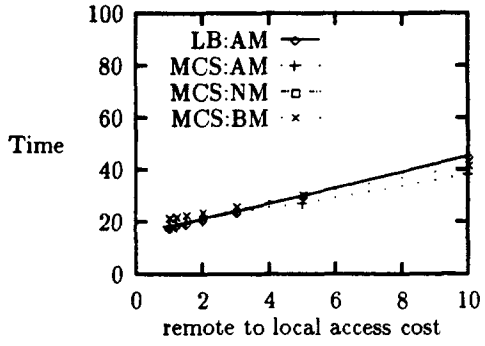

Figure 4.6: Locality Model 2.

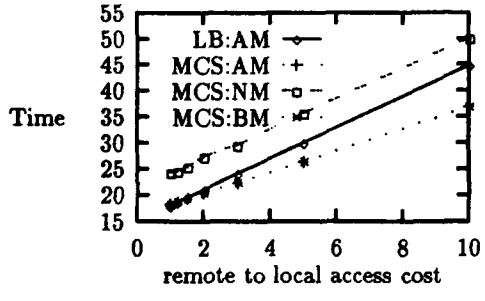

Figure 4.7: Locality Model 3.

Geometric Execution Times

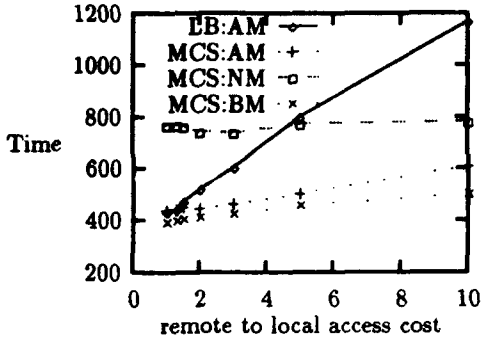

Figure 4.8: Locality Model 1.

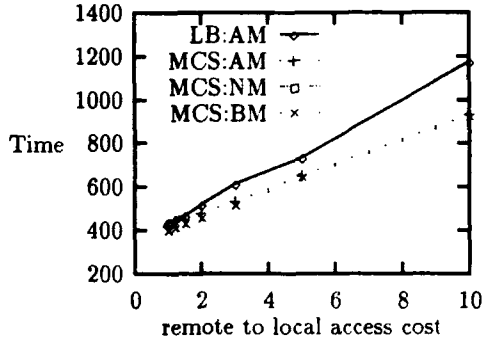

Figure 4.9: Locality Model 2.

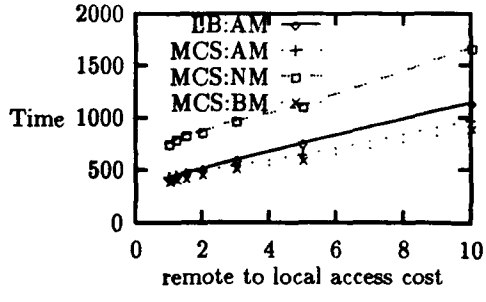

Figure 4.10: Locality Model 3.

on the Butterfly, and up to 60% on the Iris. Our experiments on the Iris show that the benefits of memory-conscious scheduling increase with the number of processors, and are particularly important when there is contention in the communication network. Our simulation results show that memory-conscious scheduling is even effective under varying distributions of data and load imbalance, and when the remote-to-local access ratio is small.

Based on our experiments, we conclude that *current multiprocessors cannot efficiently support fine-grain threads unless memory-conscious scheduling is used.* In some cases a coarse-grain decomposition out-performs a fine-grain decomposition of the same application by a factor of 3. This discrepancy cannot be attributed to thread manipulation, but is instead due to the excessive bus or network traffic associated with the fine-grain decomposition. This overhead is clearly too great a burden to justify the convenience of using fine-grain threads.

Under memory-conscious scheduling however, the performance of a fine-grain decomposition is often comparable to that of a coarse-grain decomposition for the same program. Memory-conscious scheduling removes much of the network traffic associated with the placement of fine-grain threads; the remaining difference in performance is due to the cost of creating and managing a large number of threads.

Our simulation results suggest that *memory-conscious scheduling is an appropriate compromise between policies that emphasize locality management and policies that emphasize load balancing.* In most cases memory-conscious scheduling performs better than either strict load balancing policies or strict locality management policies. Even in cases where there is a significant load imbalance, memory-conscious scheduling still manages to perform better than sophisticated load balancing policies. In the few cases where memory-conscious scheduling does not perform best, it is within 5% of the best alternative. By using initial placement to reduce the time required to execute a thread, and by dealing with load imbalance as it occurs, memory-conscious scheduling minimizes two important sources of overhead simultaneously.

5 Loop Scheduling

5.1 Introduction

Loops are the largest source of parallelism in most applications. Executing the many iterations of a loop on different processors enables applications to take advantage of parallel processors, and thereby reduce their running time. The problem of decomposing a loop into parallel tasks and executing those tasks on a multiprocessor involves finding the appropriate granularity of parallelism, so that the overhead of parallelism is kept small, while the workload is evenly balanced among the available processors. [1]

Both static and dynamic loop scheduling methods have been used to assign the iterations of a loop to processors. Static methods assign iterations to processors statically, minimizing run-time synchronization overhead. Dynamic methods defer the assignment of iterations to processors until run-time, and therefore can achieve better load balancing in the presence of unpredictable transient loads and variable execution times. The major difficulty in designing dynamic loop scheduling algorithms involves keeping the run-time synchronization overhead small, without losing the attractive load balancing properties.

The simple *static scheduling* algorithm divides the number of loop iterations among the available processors as evenly as possible, in the hope that each processor receives about the same amount of work. This algorithm minimizes run-time synchronization overhead, but does not balance the load dynamically. If all iterations do not take the same amount of time, or if processors begin executing loop iterations at different points in time, then load imbalance may arise, and will cause some processors to be idle, while other processors continue to execute loop iterations.

The simplest dynamic algorithm for scheduling loop iterations is called *self-scheduling* [65, 70]. In this algorithm, each processor repeatedly executes one iteration of the loop until all iterations are executed. The algorithm relies on a central work queue of iterations, where each idle processor gets one iteration, executes it, and repeats the same cycle until there are no more iterations to execute. Self-scheduling achieves almost perfect load balancing, since all processors finish within one iteration of each other. Unfortunately, this algorithm incurs tremendous synchronization overhead; each iteration

[1] In this chapter we consider non-nested completely parallelizable loops only. The problem of transforming nested loops into non-nested loops has been addressed previously [59].

requires atomic access to the central work queue. This synchronization overhead can quickly become a bottleneck in large-scale systems, or even in small-scale systems if the time to execute one iteration is small.

Uniform-sized chunking [41] reduces synchronization overhead by having each processor take K iterations, instead of one. This algorithm amortizes the cost of each synchronization operation over the execution time of K iterations, resulting in less synchronization overhead. Uniform-sized chunking has a greater potential for imbalance than self-scheduling however, as processors finish within K iterations of each other in the worst case. In addition, choosing an appropriate value for K is a difficult problem, which has been solved for limited cases only.

Guided Self-Scheduling [59] is a dynamic algorithm that changes the size of chunks at run-time, allocating large chunks of iterations at the beginning of a loop so as to reduce synchronization overhead, while allocating small chunks towards the end of the loop to balance the workload. Under guided-self scheduling each processor is allocated $1/P_{th}$ of the remaining loop iterations, where P is the number of processors. Assuming all loop iterations take the same amount of time to complete, guided-self scheduling ensures that all processors finish within one iteration of each other and use the minimal number of synchronization operations, even if processors start executing loop iterations at unpredictable times.

Since processors take only a small number of iterations from the work queue at the end of each loop, guided-self scheduling can suffer from excessive contention for the work queue. If each iteration takes a short time to complete, then processors spend most of their time competing to take iterations from the work-queue, rather than executing iterations. *Adaptive guided self-scheduling* [26] addresses this problem by using a back-off method to reduce the number of processors competing for iterations during periods of contention. This algorithm also avoids assigning all the time-consuming iterations to one processor, by assigning consecutive iterations to different processors, which reduces the risk of load imbalance that arises when the execution times of consecutive iterations vary widely but in a correlated fashion (e.g. if the execution time of iterations decreases linearly). As a result of these modifications, adaptive guided self-scheduling performs better than guided-self scheduling in many cases.

In some cases guided-self scheduling might assign too much work to the first few processors, so that the remaining iterations are not sufficiently time-consuming to balance the workload. This situation arises when the initial iterations of a loop are much more time-consuming than later iterations. The *factoring* algorithm [40] addresses this problem. Under factoring, allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because factoring allocates a subset of the remaining iterations in each phase, it balances load better than guided-self scheduling when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of factoring is not significantly larger than that of guided-self scheduling.

Like the factoring algorithm, the *tapering* algorithm [51] is designed for loops where the execution time of iterations varies in such a way as to cause load imbalance under

guided-self scheduling. Tapering is used for irregular loops, where the execution time of iterations varies widely and unpredictably. In the tapering algorithm execution profile information is used to estimate the average iteration time and the variance in iteration times. These estimates are used to select a chunk size that, with high probability, limits the amount of load imbalance that can occur to be within a given bound.

Although guided-self scheduling minimizes the number of synchronization operations needed to achieve perfect load balancing, the overhead of synchronization can become significant in large-scale systems with very expensive synchronization primitives. *Trapezoid self-scheduling* [73] tries to reduce the need for synchronization, while still maintaining a reasonable balance in load. This algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $\frac{N}{2P}$, and consecutive chunks differ in size $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in trapezoid self-scheduling, whereas it is a decreasing function both in guided-self scheduling and in factoring.

All of these loop scheduling algorithms attempt to balance the workload among the processors without incurring substantial synchronization overhead. Each of the algorithms assumes that an individual iteration takes the same amount of time to execute on every processor. This assumption is not valid however on many shared-memory multiprocessors. The existence of memory that is not equidistant from all processors (such as local memory or a processor cache) implies that some processors are closer to the data required by an iteration than others. Loop iterations frequently have an *affinity* [67] for a particular processor — the one whose local memory or cache contains the required data. By exploiting processor affinity, we can reduce the amount of communication required to execute a parallel loop, and thereby improve the performance.

In this chapter we describe a new loop scheduling algorithm called affinity scheduling. This algorithm attempts to balance the workload, minimize the number of synchronization operations, and exploit processor affinity. Affinity scheduling uses a deterministic assignment policy to assign repeated executions of a loop iteration to the same processor, thereby ensuring most data accesses will be to the local memory or cache. In contrast to most known algorithms, affinity scheduling employs per-processor work queues, which minimize the need for synchronization across processors. As a result of the deterministic assignment policy and per-processor work queues, affinity scheduling introduces synchronization only when load imbalance occurs. If the initial assignment of iterations to processors produces a balanced workload, all processors will finish executing at about the same time without incurring any synchronization overhead. If load imbalance occurs (i.e., a processor is idle while there are iterations to be executed), iterations migrate from one processor to another.

The next section provides the rationale for affinity scheduling, and describes the affinity scheduling algorithm. Section 5.3 presents an analytic evaluation of affinity scheduling and a comparison with other known techniques. Section 5.4 contains an experimental comparison of the known loop scheduling algorithms, based on five representative applications running on a Silicon Graphics multiprocessor workstation, and additional experiments on the BBN Butterfly and Sequent Symmetry multiprocessors.

Section 5.5 places our results in perspective, by discussing the broader issue of scheduling in shared-memory multiprocessors. Finally section 6 summarizes our results and presents our conclusions.

5.2 Affinity Scheduling

5.2.1 Rationale

Our motivation for exploiting processor affinity in loop scheduling derives from the observation that, for many parallel applications, the time spent bringing data into the local memory or cache is a significant source of overhead, ranging between 30-60% of the total execution time [16, 32, 75]. While data movement caused by true sharing is unavoidable, it is possible to minimize data movement caused by a poor assignment of iterations to processors. By scheduling a loop iteration on the processor whose local memory or cache already contains the necessary data, we can significantly reduce the execution time of the iteration.

Affinity scheduling is based on the assumption that, in many cases, loop iterations do in fact have an affinity for a particular processor. In order for this assumption to hold, it must be the case that: (1) the same data is used over and over by an iteration, and (2) the data is not removed from the local memory (or cache) before it can be reused.

Data reuse is common in many applications, particularly those that employ iterative algorithms wherein a parallel loop is nested within a sequential loop. In such cases, each iteration of the parallel loop accesses the same data on successive iterations of the enclosing sequential loop. During the first iteration of the sequential loop, each iteration of the nested parallel loop loads the required data into the local memory or cache, where it may remain during subsequent iterations of the enclosing sequential loop.

Data reuse may also occur in programs produced by a parallelizing compiler. Earlier work has suggested that nested loops be interchanged in such a way as to reduce synchronization and communication overhead [34]. The resulting loop structure nests a parallel loop within a sequential loop, again producing the desired form. If necessary, several parallel loops can be coalesced into one [58].

Whether data resides in local storage long enough to be reused is a more complicated question. Data may be removed from local storage to make room for the data needed by other iterations of the same parallel loop, or another application. If two applications share a single processor, then the data required by one application may be forced out of local storage by the other application. We can minimize this effect under time sharing by increasing the quantum, so that the time required to reload the cache is small relative to the quantum size. Even in this case, affinity scheduling will be of little help if the iterations of a parallel loop cannot be executed repeatedly within a single quantum (a distinct possibility on small-scale multiprocessors with extremely large loops). A better solution is to avoid time-sharing altogether, and employ space sharing instead, wherein each application gets some number of processors for a relatively long period of time.

Space sharing not only avoids cache (and memory) interference between applications, it also has other attractive properties that result in improved performance over time sharing [21, 21, 54] (see also chapter 3).

Even if a set of processors are dedicated to a single application, the data needed by one iteration of a loop may be evicted from local storage to make room for the data needed by another iteration of the same loop. [2] Although eviction may have been a serious problem in the past, when local caches (or memory) were quite small, it is less likely to occur in modern multiprocessors. The size of local caches and memory has grown substantially in the last few years in order to bridge the ever-widening gap between processor speeds and communication speed. For example, the BBN Butterfly offered 1 MB of local memory per node in the early 1980's, and 16 MB of memory per node in 1990. With regards to cache-coherent machines, the Sequent Symmetry (introduced around 1987) has 64 KB local caches, the Silicon Graphics 4D/480GTX (introduced around 1990) has 1 MB (second-level) local caches, and the Kendall Square Research multiprocessor (introduced in 1991) has 32 MB of coherent local memory (or cache) per processor. Given this trend, the chances are good that the local cache or memory will be large enough to hold the data for many iterations of a loop.

If eviction occurs even with very large local storage, then the program may not be suitable for execution on a multiprocessor. Efficient execution requires that a processor's working set fit in the local cache (or memory). If the working set consists of multiple iterations, and the associated data doesn't fit in local storage, then the program will thrash, spending most of its time loading data from non-local storage. This type of program will not execute efficiently on modern multiprocessors regardless of the loop scheduling algorithm in use.

Finally, there are loops that can execute efficiently on shared-memory multiprocessors, but which do not exhibit affinity. For example, a large parallel loop might force an eviction on every iteration, but if each iteration is time-consuming and makes efficient use of the local cache, then the evictions will not dominate the execution cost. Our work does not address this case; we exploit affinity only where it exists, and thereby significantly improve the performance of a large class of programs.

5.2.2 Affinity Scheduling Algorithm

We consider the loop scheduling problem to have three dimensions: load imbalance, synchronization overhead, and communication overhead due to non-local memory accesses. Our algorithm for affinity scheduling builds on previous work in loop scheduling, while also attempting to exploit processor affinity. The main ideas underlying our algorithm are:

- As with many known algorithms, we assign large chunks of iterations at the start of loop execution, so as to reduce the need for synchronization, and assign progressively smaller chunks to balance the load.

[2] We assume that the number of iterations is much larger than the number of available processors, and therefore each processor must execute multiple iterations.

- We use a deterministic assignment policy to ensure that an iteration is always assigned to the same processor. After the first execution of the iteration, that processor will contain the required data, so subsequent executions of the iteration will not need to load the data into local storage.
- We reassign a chunk to another processor (which also involves moving the required data) only if necessary to balance the load. An idle processor removes chunks from another's queue, and executes them indivisibly, so an iteration is never reassigned more than once.

We will assume that the underlying hardware or software implements a coherent memory, so that data is copied into local storage when first accessed. This copy is implemented in hardware on machines with coherent caches, such as the Symmetry and Silicon Graphics machine, and may be implemented in the operating system on machines lacking coherent caches, like the Butterfly [17, 20, 43].

Our affinity scheduling algorithm divides the iterations of a loop into chunks of size $\lceil N/P \rceil$, where N is the number of iterations in the loop, and P is the number of available processors. The i_{th} chunk of iterations is always placed on the local work queue of processor i. When a processor is idle, it removes $1/k$ of the iterations in its local work queue and executes them. [3] If a processor's work queue is empty, it finds the most loaded processor, removes $\lceil 1/P \rceil$ of the iterations in that processor's work queue, and executes them. [4]

Note that we distinguish between assigning a loop iteration to a processor's work queue, and executing the iteration on that processor. Initially, loop iterations are assigned to a processor's work queue in chunks of size $1/P$, so as to balance the load statically. Processors execute $1/k$ of the remaining iterations on their local work queue at a time, which corresponds to at most N/kP iterations. Processors execute $1/P$ of the remaining iterations from a remote work queue, which corresponds to at most N/P^2 iterations.

A pseudocode description for affinity scheduling can be found in Figure 5.1. Although we implemented this algorithm by hand for our experiments, it could easily be employed by a parallelizing compiler.

In our current implementation on small-scale multiprocessors, an idle processor examines the work queues of all the other processors and removes work from the queue with the most iterations. This implementation suffices on small-scale machines, but would not be efficient on a large-scale machine, where a scalable or randomized policy would be more appropriate [22].

There are two important differences between affinity scheduling and previous dynamic loop scheduling algorithms. First, the initial assignment of chunks to processors in affinity scheduling is deterministic. That is, processor i is always assigned the i_{th}

[3] The constant k is a parameter of our algorithm. In most of our experiments we assume k equals P. We describe the effects of changes in k in section 5.3.

[4] Synchronization is required to remove iterations from a work queue, but not to check the load on a processor.

```
loop_initialization(N,P)
// N is the number of loop iterations, P is the number of processors
{
    for(i = 0 ; i < P ; i++) {
        // assign iterations numbered ceil(i*N/P)
        // through min(N,ceil((i+1)*N/P)) to processor i
        assign_iterations(i)
    }
}

loop // executed by each processor
    // get 1/k of the local iterations to execute
    range = get_iterations_from_local_queue(1/k) ;
    if (range == empty)
        max_load = find_most_loaded_processor()  ;
        if (max_load == nil) break ;
        // get 1/P of the iterations from the most loaded processor
        range = get_iterations_from_nonlocal_queue(max_load,1/P);
        if (range == nil) break ;
        execute(range) ;
forever
```

Figure 5.1: Pseudocode for affinity scheduling

chunk of iterations to execute. For many programs, this assignment ensures that repeated executions of the loop will access data that is already stored in the local memory or cache. Second, affinity scheduling initially assumes that load imbalance will not occur, and therefore assigns the same number of iterations to each processor's work queue. Each processor gets iterations from its own local work queue; accesses to different work queues can proceed in parallel, and each access is local, and therefore cheap. If load imbalance arises, the algorithm migrates iterations from loaded processors to idle ones. Migrating iterations causes the associated data to move twice; the data must first move to an idle processor to alleviate load imbalance, and then move back to its original location to restore processor affinity. However, under affinity scheduling this overhead is introduced only when load imbalance arises, whereas other algorithms incur this overhead on every iteration.

Despite these differences, we will show that affinity scheduling has all the advantages of the best dynamic loop scheduling algorithms. That is, it balances the load dynamically, minimizes synchronization, and is immune to the arrival and departure of processors in the system.

5.2.3 Modified Factoring

The affinity scheduling algorithm is intended to address all three dimensions of the loop scheduling problem. An entirely new algorithm is not needed in order to deal with communication overhead however; previous methods can be extended to deal with this new dimension. We will now describe how to reduce the need for communication in the factoring algorithm.

During each phase of the factoring algorithm, iterations are grouped into P equal-sized chunks. Those chunks are placed in the central work queue, and each processor removes the next available chunk. Our modification to this scheme is that during each phase, processor i always removes the i_{th} chunk from the queue, rather than the chunk at the front of the queue. If the i_{th} chunk for this phase is no longer in the queue, an idle processor removes the first chunk in the queue. [5] By selecting the same chunk each time a loop executes, the modified factoring algorithm ensures that an iteration has access to the data it referenced during an earlier execution. However, each access to the central work queue is considerably more expensive than in the case of factoring or guided-self scheduling, and the additional overhead may eliminate the benefits of scheduling iterations close to their data. We will examine this issue in our experiments with the various loop scheduling algorithms.

5.3 Analytic Evaluation

Under affinity scheduling each iteration is initially assigned to a processor based on affinity considerations, and then reassigned to another processor if necessary to balance the load. Since an iteration is reassigned at most once, the algorithm is stable under

[5] As with affinity scheduling, load imbalance may cause data to move twice.

load imbalance conditions and avoids processor thrashing [68], where processors spend more time executing migrated work than executing their own assigned work.

The fact that each iteration is reassigned to another processor at most once by affinity scheduling does not imply that the number of synchronization operations associated with reassignment is linear in the number of iterations. Since iterations are assigned (and reassigned) to processors in chunks, synchronization overhead is amortized over the number of iterations in a chunk. Theorem 5.3.1 places a bound on the synchronization overhead induced by affinity scheduling.

Lemma 5.3.1 *[59] If each processor takes $1/k_{th}$ of the iterations in a work queue, the total number of accesses will be at most $k\log(N/k)$, where N is the initial number of iterations in the work queue.*

Theorem 5.3.1 *Affinity scheduling will incur at most $O(k\log(\frac{N}{Pk}) + P\log(\frac{N}{P^2}))$ synchronization operations on each work queue.*

Proof: When a processor accesses its local work queue, it removes $1/k_{th}$ of the remaining iterations. Initially, each local queue contains N/P iterations. From Lemma 5.3.1 it follows that a processor will access its own work queue at most $O(k\log(\frac{N}{Pk}))$ times. When another processor accesses that work queue, it removes $1/P_{th}$ of the remaining iterations. Again using Lemma 5.3.1 we conclude that no more than $O(P\log(\frac{N}{P^2}))$ accesses by other processors can occur. So the total number of synchronization operations to each work queue is (in the worst case) $O(k\log(\frac{N}{Pk}) + P\log(\frac{N}{P^2}))$. ∎

By way of comparison, guided-self scheduling induces $O(P\log(N/P))$ synchronization operations on the central work queue, factoring induces $O(P\log(N))$ operations, and trapezoid self-scheduling induces $4P$ operations.

One common assumption in loop scheduling is that all processors do not start executing loop iterations at the same time, as there may have been delays due to previous load imbalance or synchronization operations. Theorem 5.3.2 places a bound on the degree of imbalance that can result from using affinity scheduling under this assumption.

Lemma 5.3.2 *[59] Assume that all iterations of a loop take the same amount of time to complete. If each processor takes $1/P_{th}$ of the remaining iterations in a work queue, then all processors finish within one iteration of each other.*

Theorem 5.3.2 *Assume that all iterations of a loop take the same amount of time to complete, and that not all processors start executing loop iterations at the same time. Under affinity scheduling, all processors will finish within $\frac{N(P-k)}{P(P-1)k} + 1$ iterations of each other.*

Proof: Under affinity scheduling the worst-case imbalance occurs when all processors except one finish working on their own iterations just as the remaining processor is ready to begin working on its iterations. In this case the late processor will take $\frac{N}{Pk}$ iterations from its own

work queue, leaving $\frac{(k-1)N}{Pk}$ iterations to be divided among the other $P-1$ processors. According to Lemma 5.3.2, if each of the $P-1$ processors removes $1/P$ of the remaining iterations, they will all finish within one iteration of each other. Under this scenario, one processor will have to execute $\frac{N}{Pk}$ iterations, while each of the other processors has only $\frac{(k-1)N}{P(P-1)k}$ iterations to execute. The resulting imbalance is $\frac{N}{Pk} - \frac{(k-1)N}{P(P-1)k}$, or $\frac{N(P-k)}{P(P-1)k}$. Since processors do not, in general, start executing iterations at exactly the same time, there could be an additional disparity of one iteration. As a result, all processors will finish within $\frac{N}{Pk} - \frac{(k-1)N}{P(P-1)k} + 1$, or $\frac{N(P-k)}{P(P-1)k} + 1$ iterations from each other. ∎

Both guided-self scheduling and factoring can guarantee that all processors finish within one iteration of each other. Theorem 5.3.2 implies that if the constant k is equal to the number of processors P, then all processors will finish within one iteration of each other under affinity scheduling as well.

From these results, we see that k plays an important role in the overhead of affinity scheduling. If k is a small constant, then the number of synchronization operations per local work queue is small (proportional to $\log(\frac{N}{kP})$), while the potential for load imbalance is high (proportional to $\frac{N}{P}$). As k approaches P, affinity scheduling approaches the same worst-case load imbalance as guided-self scheduling and factoring, while simultaneously increasing the number of synchronization operations on the local work queue by a factor of P.

Selecting an optimal value for k seems a difficult task, since the best choice depends on a tradeoff between the benefits of load balancing versus the costs of synchronization. This problem is not unique to affinity scheduling however, because most loop scheduling algorithms must make this same tradeoff.

Under affinity scheduling we have separated the synchronization costs associated with access to the local work queue (as represented by k, the fraction of iterations removed from the local work queue) from the synchronization costs associated with access to remote work queues (as represented by P, the fraction of iterations removed from a remote work queue). Since synchronization operations on local work queues are usually inexpensive, we use $k = P$ in all of our implementations, which results in small initial chunks (N/P^2) and thus good load balancing properties. Smaller values of k could be used to reduce the number of accesses to local queues, while increasing the potential load imbalance.

We next consider the size of chunks of iterations that should be used with parallel loops wherein the time each iteration takes to execute is a decreasing function of the iteration index. These loops are among the most difficult to schedule because they often result in load imbalance, particularly when the scheduling algorithm assigns large chunks of loop iterations to the first few processors and successively smaller chunks to other processors. Theorem 5.3.3 indicates how many iterations each chunk should contain so that no more than $1/P$ of the remaining work is assigned to a processor at one time.

Theorem 5.3.3 *Assume a parallel loop with N iterations, where the i_{th} iteration takes time proportional to $(N-i)^k$. A chunk of size $\frac{1}{(k+1)P}$ of the iterations corresponds to*

at most $1/P_{th}$ of the remaining work to be done.

Proof: Assume that there are R more iterations to be executed. The index of the first iteration is r; the index of the last iteration is $r + R - 1$. Assume also that the time iteration x takes to complete is $c \cdot (r + R - x)^k$. This function suggests that the iteration with index r takes time $c \cdot R^k$, while the iteration with index $r + R - 1$ takes time c to complete. The total work remaining to be done in the loop is:

$$\sum_{x=r}^{r+R-1} c \cdot (r + R - x)^k$$

The time required by the first chunk of size $\frac{R}{(k+1)P}$ iterations is

$$\sum_{x=r}^{r+\frac{R}{(k+1)P}-1} c \cdot (r + R - x)^k$$

In order not to create load imbalance, we want this work to be $1/P_{th}$ of the total work to be done, or

$$\sum_{x=r}^{r+\frac{R}{(k+1)P}-1} c \cdot (r + R - x)^k = \frac{1}{P} \cdot \sum_{x=r}^{r+R-1} c \cdot (r + R - x)^k$$

Using integral approximation, we will prove the theorem by proving:

$$\begin{aligned} & \sum_{x=r}^{r+\frac{R}{(k+1)P}-1} c \cdot (r + R - x)^k \\ & \leq c \cdot R^k + \int_r^{r+\frac{R}{(k+1)P}-1} c \cdot (r + R - x)^k dx \\ & \leq \frac{1}{P} \int_r^{r+R} c \cdot (r + R - x)^k dx \qquad (5.1) \\ & \leq \frac{1}{P} \sum_{x=r}^{r+R-1} c \cdot (r + R - x)^k \end{aligned}$$

Because $(r + R - x)^k$ is a decreasing function of x we know that for all $b \geq r + 1$:

$$\sum_{x=r+1}^{b} (r + R - x)^k \leq \int_r^b (r + R - x)^k \leq \sum_{x=r}^{b-1} (r + R - x)^k \qquad (5.2)$$

These inequalities represent an upper and lower bound on the numerical value of the integral. Using (5.2) it is straightforward to prove the first and last inequality of the set of inequalities (5.1). In order to complete the proof of (5.1) we need to show that

$$c \cdot R^k + \int_r^{r+\frac{R}{(k+1)P}-1} c \cdot (r + R - x)^k dx \leq \frac{1}{P} \int_r^{r+R} c \cdot (r + R - x)^k dx$$

or

$$(k+1)R^k + (R)^{k+1} - \left(R + 1 - \frac{R}{P(k+1)}\right)^{k+1} \leq \frac{1}{P} R^{k+1}$$

or

$$\frac{k+1}{R} + 1 - \left(1 + \frac{1}{R} - \frac{1}{P(k+1)}\right)^{k+1} \leq \frac{1}{P} \tag{5.3}$$

Because $(1 \pm 1/x)^k \geq 1 \pm k/x$ and we have that

$$\frac{k+1}{R} + 1 - (1 + \frac{1}{R} - \frac{1}{P(k+1)})^{k+1} \leq \frac{k+1}{R} + 1 - 1 - \frac{k+1}{R} + \frac{1}{P} \leq \frac{1}{P}$$

which proves inequality (5.3), which in turn proves inequality (5.1). ∎

Theorem 5.3.3 suggests that when all iterations take the same amount of time, $1/P_{th}$ of the iterations corresponds to $1/P_{th}$ of the workload. When the iterations have a decreasing triangular form, that is iteration i takes time proportional to $(N - i)$, then $1/(2P)_{th}$ of the iterations corresponds to $1/P_{th}$ of the workload. When the iterations have a decreasing parabolic form, that is iteration i takes time proportional to $(N - i)^2$, then $1/(3P)_{th}$ of the iterations corresponds to $1/P_{th}$ of the workload.

Loops with decreasing workloads, such as those described above, are among the most difficult loops to schedule. The scheduling algorithm must be careful to avoid assigning so many iterations to one processor that the remaining iterations are insufficient to balance the workload. Our experiments indicate that the factoring and trapezoid algorithms have better load balancing properties than guided-self scheduling for this type of loop. This result can be traced to the fact that both factoring and trapezoid start with a chunk that contains $1/(2P)_{th}$ of the iterations to be scheduled, while guided-self scheduling starts with a chunk that contains $1/P_{th}$ of the iterations. According to theorem 5.3.3, the first chunk will be the bottleneck in guided-self scheduling, while it will not be a bottleneck for factoring or trapezoid.

In general, if a loop scheduling algorithm assigns less than $1/P_{th}$ of the remaining workload to each idle processor, then the minimum imbalance will result. Theorem 5.3.3 states how many iterations correspond to this fraction of the remaining workload for loops wherein successive iterations require a polynomially decreasing amount of work. If the amount of work per iteration increases polynomially, then the loop is easy to schedule: $1/(kP)_{th}$ of the remaining iterations *always* corresponds to less than $1/P_{th}$ of the remaining work.

Summarizing our results, affinity scheduling (with $k = P$) offers worst-case load imbalance guarantees that are the the same as (or in some cases better than) those of guided-self scheduling and factoring, but can, in the worst case, introduce about P times more synchronization operations. Fortunately, these synchronization operations are directed to P different work queues, and so the number of *serialized* synchronization operations under affinity scheduling is somewhat smaller than the number of serializable synchronization operations under guided-self scheduling or factoring. Since affinity scheduling can also dramatically reduce communication overhead, affinity scheduling should perform much better than either guided-self scheduling or factoring. We will now examine the relative performance of these loop scheduling algorithms experimentally.

5.4 Experimental Evaluation

In order to evaluate the performance benefits of affinity scheduling, we implemented many of the known loop scheduling methods by hand on a Silicon Graphics 4D/480GTX Iris workstation, a bus-based, cache-coherent machine with 8 processors. We then measured the performance of each of the scheduling algorithms on a suite of applications.

5.4.1 Scheduling Algorithms

We implemented the following loop scheduling algorithms by hand on the Iris: static scheduling (STATIC), self-scheduling (SS), guided-self scheduling (GSS), factoring (FACTORING), trapezoid self-scheduling (TRAPEZOID), affinity scheduling with $k = P$ (AFS), modified factoring (MOD-FACTORING), and a hand-optimized algorithm (BEST-STATIC). BEST-STATIC represents our attempt at the best static assignment possible, given complete knowledge of the application and its input. We implemented this assignment by hand, after examining the application and the input, so as to maximize locality of reference and minimize load imbalance. While not generally realizable, since it requires programmer intervention and assumes knowledge of the application's input, BEST-STATIC is a useful base-line for evaluating other loop scheduling algorithms.

5.4.2 Applications

We carefully selected five application programs that present loop scheduling algorithms a range of opportunities for addressing load imbalance, synchronization overhead, and communication overhead. Our application suite contains the following programs:

- Successive Over-Relaxation (SOR):

```
DO SEQUENTIAL 19 I = 1,MAXITERATIONS
    DO PARALLEL 29 J = 1,N
        DO SEQUENTIAL 39 K = 1,N
           A(J,K) = UPDATE(A,J,K)
        39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

  All iterations of the parallel loop take about the same time to execute, so better load balancing algorithms are not likely to produce much better performance. However, the i_{th} iteration of the parallel loop always accesses the i_{th} row of the matrix, so scheduling algorithms that exploit processor affinity are likely to produce much better performance.

- Gaussian Elimination:

```
DO SEQUENTIAL 19 K = 2,N
    DO PARALLEL 29 I = K,N
        DO SEQUENTIAL 39 J = K-1,N+1
            A[I][J] = A[I][J] - A[K-1][J] * A[i][K-1]/A[K-1][K-1]
        39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

This application exhibits some load imbalance across iterations, but it also offers many opportunities for exploiting processor affinity. Although successive executions of an iteration of the parallel loop do not access exactly the same matrix elements each time, there is significant overlap in the elements referenced by successive executions of an iteration. As with SOR, we expect scheduling algorithms that exploit affinity to improve performance substantially.

- Transitive Closure:

```
DO SEQUENTIAL 19 K = 1,N
    DO PARALLEL 29 J = 1,N
        IF (A(J,K) .EQ. TRUE)  THEN
            DO SEQUENTIAL 39 I = 1,N
                IF (A(K,I) .EQ. TRUE) A(J,I) = TRUE
            39 CONTINUE
    29 CONTINUE
19 CONTINUE
```

The distinguishing characteristic of this application is that each iteration of the parallel loop may take time $O(1)$ or $O(N)$ (where the input matrix is of size $N \times N$), depending on the input data. Since the input values determine the variation in iteration execution time, this application will serve to evaluate the effectiveness of load balancing for each scheduling algorithm. This application will also benefit from some form of affinity scheduling, since the i_{th} iteration of the parallel loop always accesses the i_{th} row of the matrix.

- Adjoint Convolution:

```
DO PARALLEL 19 I = 1,N*N
    DO SEQUENTIAL 29 K = I,N*N
        A(I) = A(I) + X*B(K)*C(I-K)
    29  CONTINUE
19  CONTINUE
```

This application exhibits significant load imbalance; the i_{th} iteration of the parallel loop takes time proportional to $O(n^2 - i)$. There is no affinity to exploit however, so this application serves to evaluate the effectiveness of load balancing in the absence of affinity.

```
DO SEQUENTIAL 1 I1 = 1,50
     DO PARALLEL 2 I2 = 1,10
         DO PARALLEL 3 I3 = 1,10
             DO PARALLEL 4 I4 = 1,10
                 {10}
                 [if C then {50}]
             4 CONTINUE
         3 CONTINUE
     2 CONTINUE
     DO PARALLEL 5 I5 = 1,100
         {50}
         DO PARALLEL 6 I6 = 1,5
             {100}
             [if C then {30}]
         6 CONTINUE
     5 CONTINUE
     DO PARALLEL 7 I7 = 1,20
         DO PARALLEL 8 I8 = 1,4
            {30}
         8 CONTINUE
     7 CONTINUE
 1 CONTINUE
```

Figure 5.2: Structure of the L4 application

- L4: This application was used as a benchmark in [59]; we include it in our study for comparison with previously published results. The structure of L4 can be found in Figure 5.2. L4 is an example of a hybrid application with non-perfectly nested and multi-way nested parallel loops. In our experiments, all **if** statements are true with probability 0.5. As this application does not perform any memory accesses, there is no affinity to be exploited.

Table 5.1 summarizes the properties of our application suite with respect to load imbalance and affinity. If an application exhibits load imbalance, the iterations of the loop may take varying amounts of computation time, so a static scheduling algorithm may not be appropriate. If an application exhibits affinity, we can improve performance by scheduling iterations appropriately.

5.4.3 Comparison of Loop Scheduling Algorithms

In this section we compare the performance of the various loop scheduling algorithms using the application suite. Due to the large number of scheduling algorithms we consider, we will represent algorithms with comparable performance with a single line in the performance graphs.

Application	Load imbalance	Affinity
SOR	none	yes
Gauss elimination	little	yes
Transitive closure	input dependent	yes
Adjoint convolution	large	no
L4	little	no

Table 5.1: Load imbalance and affinity characteristics of the application suite.

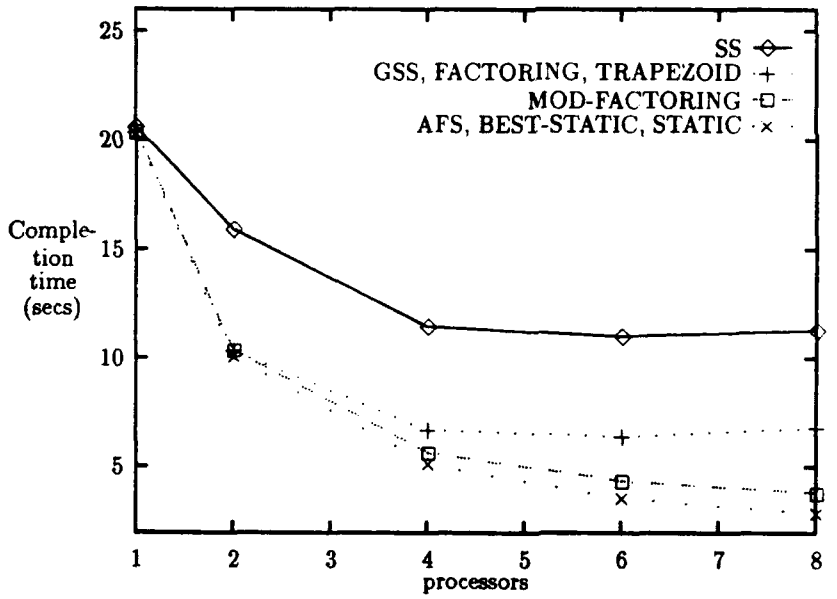

Figure 5.3: Performance of loop scheduling algorithms for SOR.

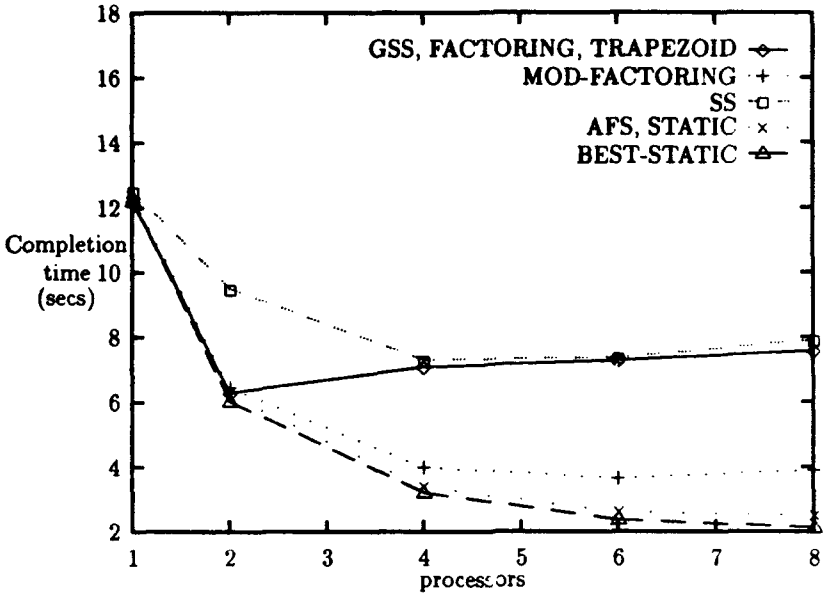

Figure 5.4: Performance of loop scheduling algorithms for Gaussian elimination.

Figure 5.3 presents the completion time (in seconds) of the SOR program ($N = 512$) running on 1 to 8 processors. As can be seen in the figure, SS performs the worst of all, due to its high synchronization overhead. Other algorithms with lower synchronization overhead, such as GSS, FACTORING, and TRAPEZOID, perform much better than SS — since there is no significant difference in the execution time of iterations, sophisticated load balancing schemes aren't necessary for this application. All of these algorithms perform worse than the algorithms that exploit affinity. Both STATIC and AFS are comparable to the best possible static algorithm. MOD-FACTORING lies between AFS and FACTORING, since it requires less communication than FACTORING, but requires more expensive access to the work queue than AFS. These results confirm that affinity scheduling can improve the performance of loop scheduling algorithms.

Figure 5.4 plots the completion time of the Gaussian elimination program ($N = 384$) under the different scheduling algorithms. It is surprising to see that none of the scheduling algorithms that ignore processor affinity can effectively utilize more than two processors. There is simply too much contention for the shared bus under these algorithms, since every iteration must load data into the local cache. SS performs worst of all, because of its high synchronization overhead, but the performance difference narrows quickly as the communication costs of GSS, FACTORING, and TRAPEZOID start to dominate synchronization costs. Once again, AFS and STATIC perform the

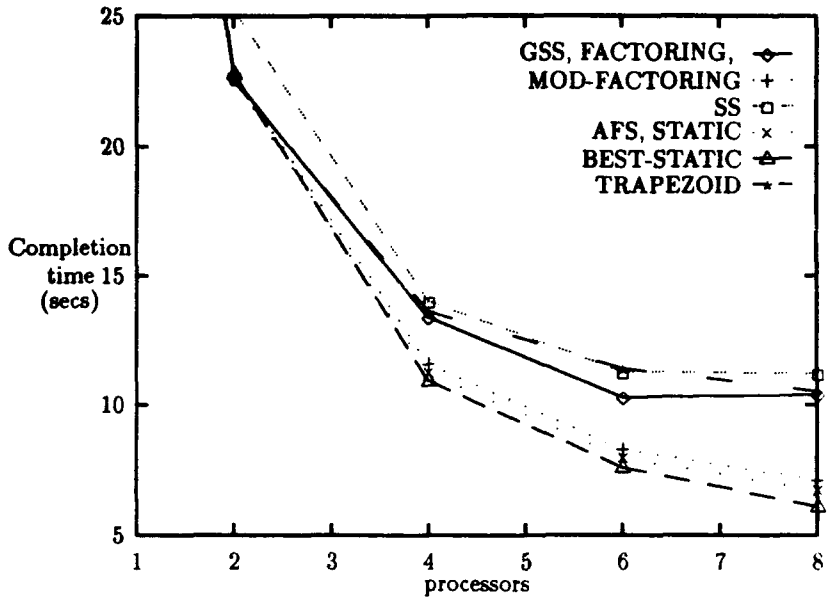

Figure 5.5: Performance of loop scheduling algorithms for transitive closure (random input).

best; they are within 10% of BEST-STATIC in the worst case, and a factor of 3 better than the traditional dynamic loop scheduling algorithms. In addition, both AFS and STATIC can effectively use all 8 processors.

This application is a perfect example of the fact that the dominant source of overhead in many applications is communication (caused by cache misses), not synchronization. Loop scheduling algorithms that focus on synchronization overhead alone perform poorly when compared to algorithms that reduce communication overhead by exploiting processor affinity.

Figure 5.5 presents the completion time of the transitive closure program when given a random input graph of 512 nodes, with about 8% of the edges present. Because the load is averaged over all iterations, preserving affinity takes precedence over load balancing. As a result, AFS, STATIC, and MOD-FACTORING perform better than GSS, FACTORING, SS, and TRAPEZOID.

Figure 5.6 presents the completion time of the transitive closure application when given a skewed input graph of 640 nodes containing a clique of 320 nodes, and no other edges. This is the first example where there is significant imbalance in the computation across iterations, which explains why STATIC performs poorly. Although SS manages to balance the load, it still suffers from high synchronization overhead. The surprising

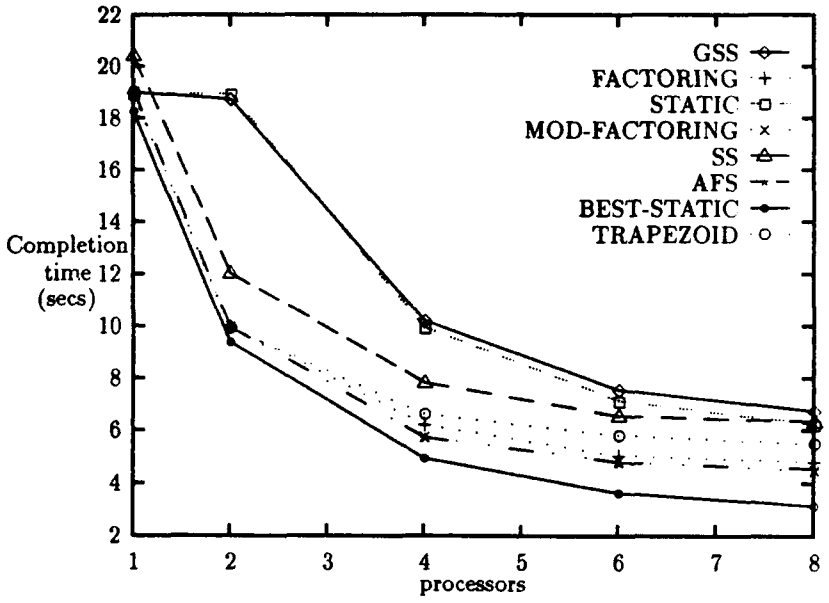

Figure 5.6: Performance of loop scheduling algorithms for transitive closure (skewed input).

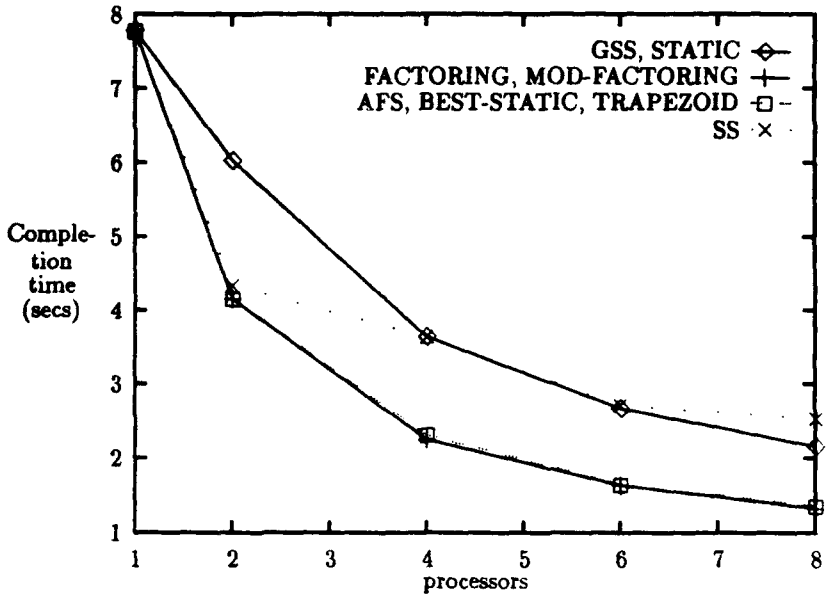

Figure 5.7: Performance of loop scheduling algorithms for adjoint convolution.

result in Figure 5.6 is that GSS performs worst of all. Although GSS assigns only $1/P$ of the iterations to the first processor, those iterations contain $2/P_{th}$ of the total work; the remaining iterations do not have enough work to balance the load. Both FACTORING and TRAPEZOID start with a smaller initial chunk of iterations, and therefore balance the load better. AFS and MOD-FACTORING have the same load balancing properties as FACTORING and TRAPEZOID, but exploit affinity as well.

Although AFS and MOD-FACTORING perform the best, the improvement over FACTORING and TRAPEZOID is not greater than 15%. The existence of significant load imbalance forces an affinity scheduler to override the initial assignment of iterations to processors and instead execute iterations on any available idle processor. Each time an iteration moves to another processor, the data must be loaded into a different cache. This is also why AFS does not perform as well as BEST-STATIC, which has knowledge of the input, and is therefore able to distribute the clique nodes evenly among the processors, while maintaining processor affinity.

Figure 5.7 presents the performance of the scheduling algorithms for the adjoint convolution program with $N = 75$. In this application, iterations have no affinity for a particular processor, since the parallel loop is not embedded within a sequential loop. There is significant load imbalance across iterations however, since the first iteration takes time proportional to $O(N^2)$, while the last iteration takes time proportional to

$O(1)$. As expected, loop scheduling algorithms that emphasize load balancing, such as FACTORING, MOD-FACTORING, TRAPEZOID and AFS, perform the best. GSS and the static methods assign too much work to the first few processors, and suffer load imbalance as a result. SS again suffers from high synchronization overhead. These results are consistent with those of [40].

We should note that a trivial change to our implementation of GSS would improve its performance to be comparable to FACTORING, although not as good as AFS for these examples. Instead of taking $\lceil N/P \rceil$ iterations, each processor could take $\lceil N/(kP) \rceil$ iterations, where k is an appropriate constant. With this change, GSS could start with smaller chunks, leaving more opportunities to balance the load, without introducing significant synchronization overhead. Eager and Zahorjan [26] argue that decreasing the chunk size is not enough to balance the load if the execution time of iterations decreases at a fast enough rate. Theorem 5.3.3 quantifies this relationship between the variance in iteration execution times and the resulting load imbalance, and suggests that if the rate of decrease is polynomial with exponent k, and each processor takes no more than $\frac{1}{(k+1)P}$ of the remaining iterations, no imbalance will occur. Thus, a simple decrease in the chunk size is probably enough to balance the load for nearly all programs.

Load imbalance is particularly important in the adjoint convolution problem because the computation times of the iterations decrease linearly; the first few chunks could become a bottleneck. Rather than decrease the chunk size at the beginning of the loop, we could schedule the loop backwards, so that the last iterations execute first. (Reverse execution works in this case because there are no dependencies among the iterations.) Figure 5.8 presents the performance of several loop schedulers on the adjoint convolution problem, when scheduling the iterations in reverse order. We see that all scheduling algorithms (apart from SS) perform reasonably well, and are comparable in performance to the best scheduling algorithms that execute the loop iterations in index order. Although executing the iterations in reverse order may increase the potential load imbalance (since the last iterations to be executed are the most time-consuming), the potential imbalance is a negligible percentage of the total completion time of the application. If there are N iterations, and the i_{th} iteration takes $O(N - i)$ time to execute, the last iteration to be executed under reverse ordering takes about $O(N)$ time, while the total completion time of the application is about $O(N^2/P)$. Thus, the potential imbalance (time $O(N)$) is asymptotically small when compared to the total completion time ($O(N^2/P)$), unless the number of processors is on the order of the number of loop iterations.

Finally, figure 5.9 plots the performance of the loop scheduling algorithms for the L4 application. Since there are no memory references in L4, we would not expect an affinity scheduler to perform any better than a scheduler that ignores affinity. In fact, all loop schedulers perform about the same, although the dynamic schedulers perform a bit better than the static scheduler, and self-scheduling is clearly the worst. These results for L4 are consistent with those reported in [59].

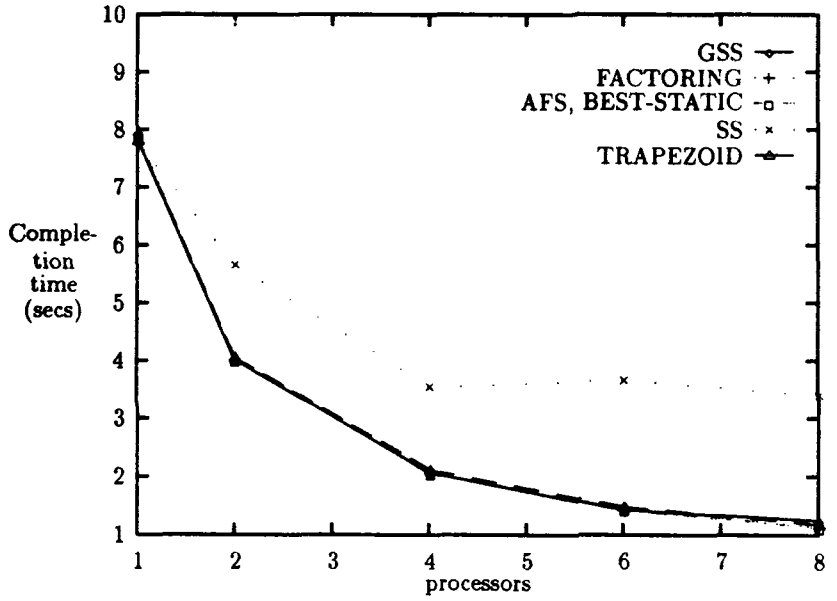

Figure 5.8: Performance of loop scheduling algorithms for adjoint convolution (reverse index scheduling).

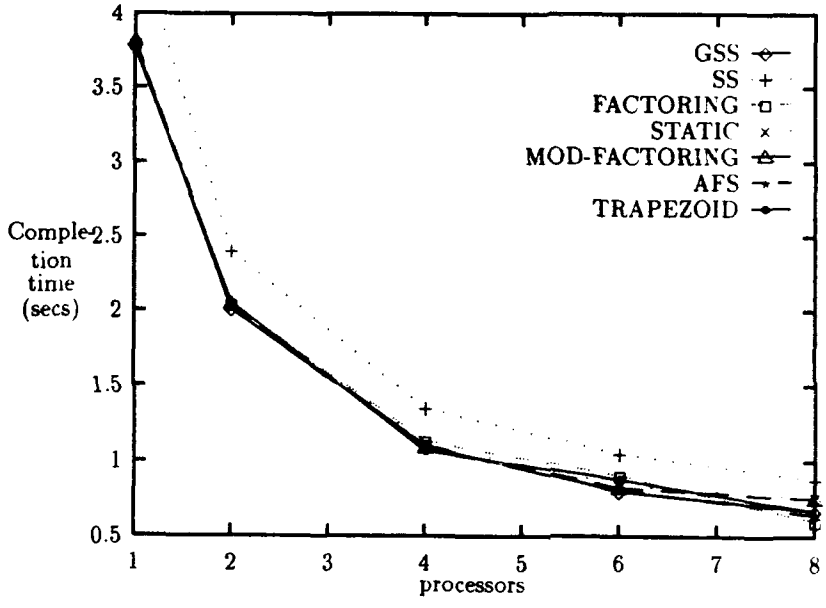

Figure 5.9: Performance of loop scheduling algorithms for application L4.

5.4.4 Effects of Load Imbalance

In order to explore the effect of load imbalance in isolation, we implemented three dynamic loop scheduling algorithms (AFS, GSS, and TRAPEZOID) by hand on the BBN Butterfly. None of our loop scheduling algorithms on the Butterfly preserve affinity, and even the distributed work queues require non-local access, so any performance differences can be attributed to the load balancing properties of the various algorithms.

We executed three applications on the Butterfly, while progressively introducing more imbalance in the computation. The first application has the following form:

```
DO PARALLEL 19 I = 1, N
   DO SEQUENTIAL 29 J = 1, N-I
      COMPUTE
   29 CONTINUE
19 CONTINUE
```

This application is similar to adjoint convolution, in that the first few iterations of the parallel loop have much more work to do than the last few iterations. Figure 5.10 plots the performance of the three loop scheduling algorithms for this application, where $N = 5000$. AFS and TRAPEZOID have comparable performance, and both perform better than GSS. The reason for this is given by theorem 5.3.3, which states that the workload of this application is evenly balanced when processors take $1/(2P)$ of the remaining iterations. TRAPEZOID starts with chunks of exactly that size, while AFS uses smaller chunks, which results in slightly greater synchronization overhead.

Our second application has even greater load imbalance: iteration i takes time proportional to $(N - i)^2$.

```
DO PARALLEL 19 I = 1, N
   DO SEQUENTIAL 29 J = 1, (N-I)**2
      COMPUTE
   29 CONTINUE
19 CONTINUE
```

According to theorem 5.3.3, each processor should take $1/(3P)$ of the remaining iterations to balance the load evenly. TRAPEZOID allocates chunks larger than that, but smaller than the chunks used by GSS. Therefore, we would expect TRAPEZOID to behave worse than AFS, but better than GSS. Figure 5.11 plots the results for this program, with $N = 200$. As expected, AFS performs better than TRAPEZOID, which performs better than GSS. Note however that TRAPEZOID is very close to AFS when the number of processors is close to 50 and $N = 200$. Theorem 5.3.3 explains why: given 50 processors, the first chunk allocated by TRAPEZOID is of size $200/(2 * 50) = 2$ iterations, while the maximum number of iterations that can be allocated without creating imbalance according to theorem 5.3.3 is $200/(3 * 50) = 1.5$ iterations. Thus, TRAPEZOID is within one iteration of the optimal allocation, which in practice gives performance comparable to AFS.

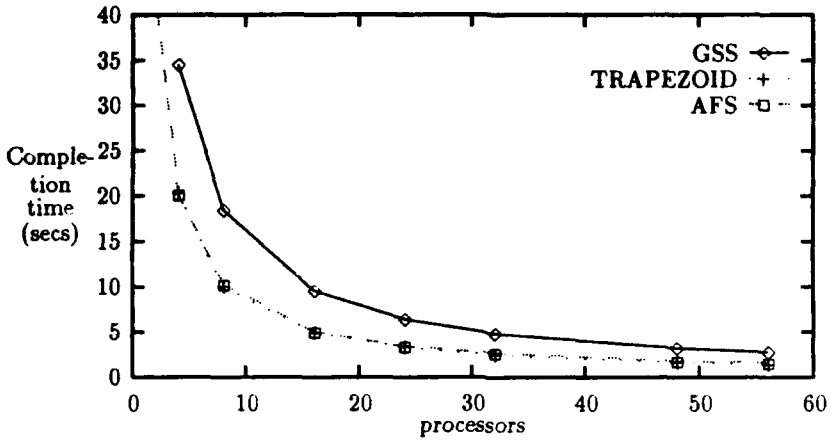

Figure 5.10: Performance of loop scheduling algorithms on the Butterfly under triangular workload.

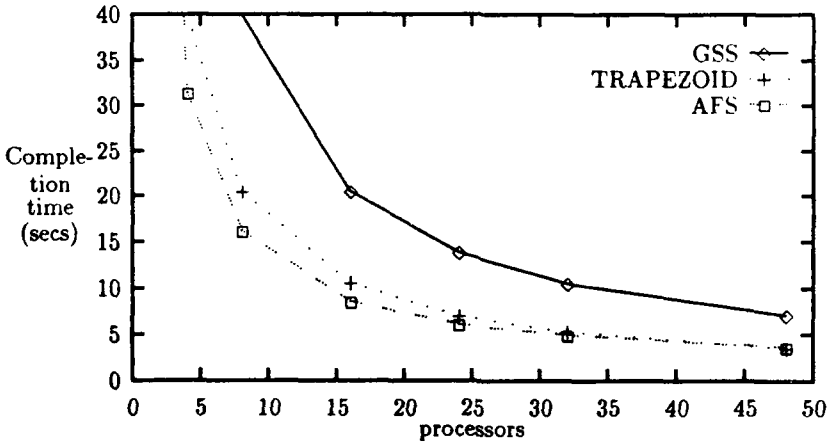

Figure 5.11: Performance of loop scheduling algorithms on the Butterfly under decreasing parabolic workload.

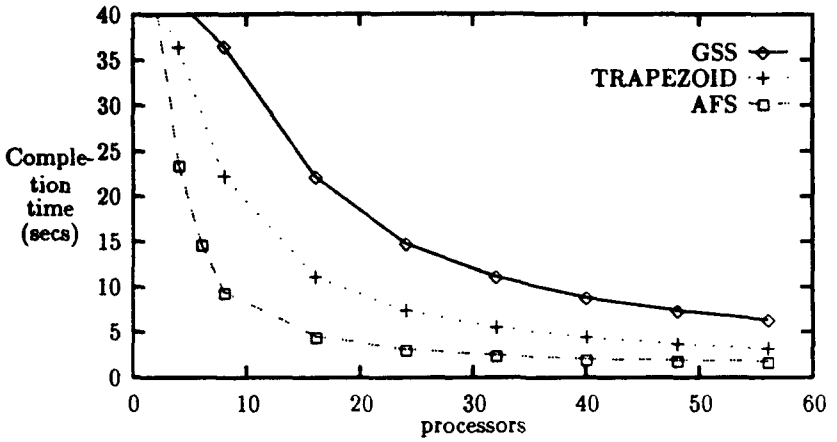

Figure 5.12: Performance of loop scheduling algorithms on the Butterfly when load is in first 10% of iterations.

Our final application has imbalance comparable to that of transitive closure. That is, the first 10% of the iterations take 100 time units to complete, while the remaining 90% of the iterations take one time unit to complete. The code for the application is:

```
DO PARALLEL 19 I = 1,N
    IF (I.LT.(N/10))  THEN
        COMPUTE(100)
    ELSE
        COMPUTE(1)
    ENDIF
19 CONTINUE
```

If the first processor takes more than $1/(10P)$ of the iterations, it will get more than $(1/P)_{th}$ of the work, and will therefore be the last processor to finish. Figure 5.12 plots the results of executing this program on the Butterfly with $N = 50000$. In this figure, AFS is clearly superior to TRAPEZOID and GSS. Both GSS and TRAPEZOID can be improved, at the expense of synchronization overhead, by starting with smaller chunks of iterations. AFS can afford to start with small chunks of iterations because it uses a distributed work queue, which results in either smaller synchronization overhead for the same load balancing properties, or comparable synchronization overhead for superior load balancing properties.

Delay	GSS	TRAPEZOID	FACTORING	AFS	
				$k = 2$	$k = P$
0.0625N	2.31	2.34	2.31	2.42	2.32
0.125N	2.44	2.44	2.45	2.59	2.44
0.1875N	2.53	2.54	2.52	2.82	2.52
0.2031N	2.54	2.57	2.54	2.68	2.54
0.2187N	2.58	2.6	2.58	2.58	2.58
0.25N	2.9	2.9	2.9	2.9	2.9

Table 5.2: Execution time (in seconds) of simple, balanced loop program with non-uniform start times.

5.4.5 Effect of Processor Arrival Time

In our previous examples, we assumed that all processors begin executing iterations at about the same time. We now focus on the case where not all processors start executing loop iterations at the same time. Static loop scheduling algorithms are clearly inappropriate for this situation; robust dynamic loop scheduling algorithms should be able to distribute the load evenly independent of the starting time of processors.

According to the analysis in section 5.3, if all iterations take the same time to complete, then under guided-self scheduling, factoring, and affinity scheduling (with $k = P$), all processors finish within one iteration of each other, regardless of the starting time of processors. To confirm this fact experimentally, we implemented a simple, balanced parallel loop with 200 million iterations and no memory accesses on the Iris. Since our loop has no affinity to exploit, the performance differences among the algorithms can be attributed to any load imbalance caused by the non-uniform starting time of processors.

In this set of experiments, all processors start executing loop iterations at the same time, except for one processor, which is delayed for time t_1. We varied the delay and measured the execution time of our simple loop under the different scheduling algorithms. The results appear in table 5.2. In the table, the delay column represents the number of iterations one processor was delayed. For example, in the first experiment, a processor is delayed for the amount of time it takes one processor to execute one-sixteenth of the iterations. Within this time, the other seven processors can execute $7/16 = 0.43$ of the iterations.

The measured results show that all algorithms perform about the same in the presence of non-uniform starting times for all processors. These results are as expected for GSS, FACTORING and AFS (with $k = P$), because each of these scheduling algorithms guarantees that all processors finish within one iteration of each other. TRAPEZOID also performs close to the best algorithm in each case. Not surprisingly, AFS with $k = 2$ performs worst of all, but even this algorithm is within 10% of the best algorithm.

These experiments suggest that having processors with different arrival times does not affect the performance of good loop scheduling algorithms. The maximum imbalance introduced depends on the size of allocations of iterations relative to the number of

remaining iterations. If the remaining iterations are enough to balance the work evenly (as is the case in most loop scheduling algorithms), then different arrival times do not impose any noticeable overhead. The two factors that distinguish the performance of the various loop scheduling algorithms in our experiments are the load imbalance inherent in the computation, and the ability to preserve affinity in the scheduler.

5.4.6 Synchronization Overhead

In this section we focus on the synchronization overhead imposed by each scheduling algorithm, so as to verify experimentally our analytic results, and to quantify the synchronization overhead incurred by our application suite. Our metric for synchronization overhead is the number of times a processor removes iterations from a work queue. The time required to remove iterations from a work queue might be a more accurate metric, but we are primarily interested in the number of synchronization operations required by each algorithm, and not the implementation details of a particular algorithm on a particular machine.

Every algorithm except affinity scheduling uses a central work queue, wherein each access to the work queue is a global synchronization operation. For affinity scheduling, we identify separately the number of operations performed on local work queues from the operations performed on remote work queues. We note however that on many architectures, operations on remote queues under affinity scheduling would be cheaper than global synchronization operations on a central work queue, since there is less contention for access to each distributed work queue under affinity scheduling.

We should also note that load imbalance does not affect the number of synchronization operations performed by SS, GSS, FACTORING and TRAPEZOID. Load imbalance does affect the number of synchronization operations performed by AFS however, because AFS responds to imbalance dynamically by migrating iterations. Thus, the number of remote synchronization operations performed by AFS will give us insight into the migration overhead incurred by the algorithm.

We will use SOR as an example of a well-balanced application, and adjoint-convolution and transitive closure (with a skewed input) as examples of applications with considerable variance in computation times across iterations. In the transitive closure example, all the work is contained in the first half of the iterations, while in the adjoint convolution example, the computation times of the iterations are linearly decreasing.

Table 5.3 shows the number of synchronization operations per loop incurred by SOR under the various scheduling algorithms. In our example, there are 512 iterations per loop, so self-scheduling (SS) induces exactly 512 synchronization operations, regardless of the number of processors. TRAPEZOID requires the smallest number of synchronization operations, followed by GSS and FACTORING. As expected, AFS requires a very small number of costly remote synchronization operations, and induces about as many local synchronization operations per queue as TRAPEZOID.

Note that all of the entries in table 5.3 represent an integer number of synchronization operations, except the entries for affinity scheduling. There are two reasons for this.

First, the individual entries in the table for affinity scheduling represent an average across all processors for local and remote synchronization operations. Second, repeated executions of the same parallel loop under affinity scheduling do not always require the same number of local and remote synchronization operations.

Table 5.4 shows the number of synchronization operations per loop incurred by transitive closure (with a skewed input matrix) under the various scheduling algorithms. Once again, SS induces a large number of synchronization operations independently of the number of processors. TRAPEZOID requires the fewest serialized synchronization operations, but is only slightly better than AFS.

Even though the input matrix causes a large load imbalance in this application, AFS requires only one or two remote synchronization operations per work queue to balance the load. Each processor accesses a local work queue 20-30 times on average, but rarely accesses a remote work queue. Whereas traditional loop scheduling algorithms always access non-local work queues, AFS accesses a non-local work queue only 5-10% of the time, and yet balances the load just as well. On machines where access to a local work queue is much cheaper than access to a remote work queue (either due to the cost of non-local access or the cost of non-local synchronization primitives), this property of affinity scheduling could have enormous performance advantages.

Table 5.5 presents the total number of synchronization operations for the adjoint convolution application under the various scheduling algorithms. TRAPEZOID again has the smallest number of synchronization operations. Although AFS does more synchronization operations than TRAPEZOID, the additional overhead is not noticeable, because synchronization is relatively inexpensive on the Iris multiprocessor, and because the number of processors we used in our experiments was rather small. In both cases synchronization was less than 1% of the execution time, so any small savings in the number of synchronization operations would have almost no impact on total execution time.

To confirm that synchronization overhead is not an important factor in the comparative performance of loop scheduling algorithms on shared-memory multiprocessors, we implemented a simple, balanced parallel loop on the Butterfly. In our implementation of affinity scheduling on the Butterfly, all work queues require non-local access. Since our loop has no affinity to exploit, the performance differences among the algorithms can be attributed to synchronization overhead. The results appear in Figure 5.13. As can be seen from the figure, GSS, TRAPEZOID, and AFS have comparable performance when the effects of affinity scheduling, distributed work queues, and load imbalance are factored out.

5.4.7 Architectural Trends and Affinity Scheduling

Our experiments on the Iris confirm that communication overhead is a dominant factor in application performance on modern shared-memory multiprocessors. Why then do so many of the known loop scheduling algorithms ignore communication overhead? Because the hardware trends discussed in chapter 2 have produced a qualitative change in the relative cost of communication and computation in the last few years.

Processors	SS	GSS	FACTORING	TRAPEZOID	AFS (per work queue)	
					remote	local
1	512	1	10	3	0	1
2	512	10	18	7	0.5	7.3
4	512	23	32	13	1.0	16.8
6	512	33	50	16	1.1	21.8
8	512	43	56	27	0.4	27

Table 5.3: Number of synchronization operations for SOR (N= 512).

Processors	SS	GSS	FACTORING	TRAPEZOID	AFS (per work queue)	
					remote	local
1	640	1	11	3	0	1
2	640	11	20	7	1.5	8.4
4	640	23	36	13	2.1	16.8
6	640	34	52	18	1.1	23.8
8	640	45	64	22	0.7	28.3

Table 5.4: Number of synchronization operations for transitive closure on a skewed 640-node graph.

Processors	SS	GSS	FACTORING	TRAPEZOID	AFS (per work queue)	
					remote	local
1	5625	1	14	3	0	1
2	5625	14	29	7	4	11
4	5625	31	49	14	6.75	20.2
6	5625	46	69	21	8.3	29.3
8	5625	61	89	28	9.87	35.6

Table 5.5: Number of synchronization operations for adjoint convolution N = 75.

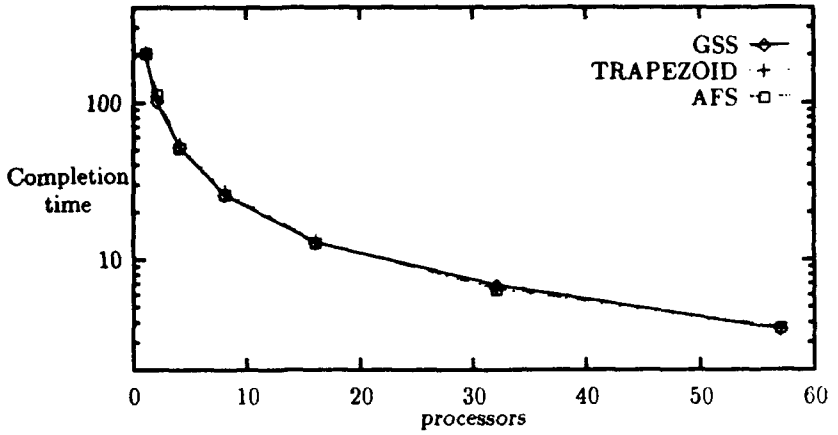

Figure 5.13: Performance of loop scheduling algorithms on the BBN Butterfly.

In order to demonstrate the impact of this change on loop scheduling, we executed our Gaussian elimination program on a Sequent Symmetry S81 multiprocessor, a bus-based, cache-coherent machine that predates the Iris. The processors on the Iris are about 30 times faster than the processors on the Symmetry, but the peak bandwidth of the Symmetry bus is 80 MB/sec, while the peak bandwidth of the Iris bus is only 64 MB/sec. Figure 5.14 plots the execution time of Gaussian elimination on a 256 by 256 matrix under three dynamic loop scheduling algorithms on the Symmetry. From this figure we can see that AFS and GSS are comparable in performance on the Symmetry, while our earlier results showed that AFS clearly dominates GSS on the Iris. We can conclude that the ability of AFS to exploit processor affinity in Gaussian elimination is of little value on the Symmetry, since communication is cheap relative to computation.

We also see in figure 5.14 that TRAPEZOID performs 10-15% worse than both AFS and GSS on this application. The cause of this disparity can be traced to the load balancing properties of TRAPEZOID. When all iterations take the same time to execute, processors finish within one iteration of each other under guided-self scheduling [59]. Under TRAPEZOID, processors finish within several iterations of each other [73]. When an iteration takes a long time to complete, the imbalance introduced by the trapezoid algorithm can be noticeable. Although the trapezoid algorithm requires fewer accesses to the work queue, the Sequent does not employ a large number of processors, and therefore the low synchronization overhead of TRAPEZOID does not outweigh the load imbalance it causes.

These results suggest that communication was a relatively minor issue on the previous generation of shared-memory multiprocessors, and that both load imbalance and

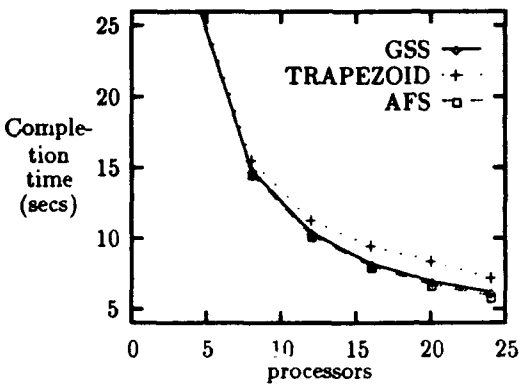

Figure 5.14: Gaussian elimination on the Sequent Symmetry.

synchronization overhead were dominant. Our results on the Iris argue that the situation has changed dramatically, so much so that communication is now the dominant factor in performance.

5.4.8 Scalable Multiprocessors

To demonstrate that the affinity effects we measured on the Iris hold and are even stronger on recent large-scale multiprocessors, we performed several experiments on the KSR-1, a large-scale cache-coherent multiprocessor, released in 1992, two years after the Iris. We are interested in investigating if the architecture trends we observed among the Sequent and the Iris still hold for the KSR-1. Our experiments use those programs from our application suite that have some locality to preserve, which includes Gaussian elimination, SOR, and transitive closure.

Figure 5.15 presents the completion time of Gaussian elimination on a 1024 by 1024 matrix under various loop scheduling algorithms on the KSR-1. We see that AFS is best of all algorithms. It improves the completion time of the application by a factor of 3.7 compared to FACTORING and GSS, and by a factor of 2.8 compared to TRAPEZOID. TRAPEZOID outperforms both GSS and FACTORING because it has the smallest number of synchronization operations, and synchronization is very expensive on the KSR, far more expensive than on the Iris (which has hardware locks).

MOD-FACTORING exhibits interesting behavior. It behaves reasonably well on a small number of processors, somewhere between AFS and TRAPEZOID. For larger numbers of processors, its behavior gets increasingly unstable, and starts to approach the performance of FACTORING very quickly. The reason for this sudden change is that large numbers of processors can easily introduce small amounts of imbalance. Such

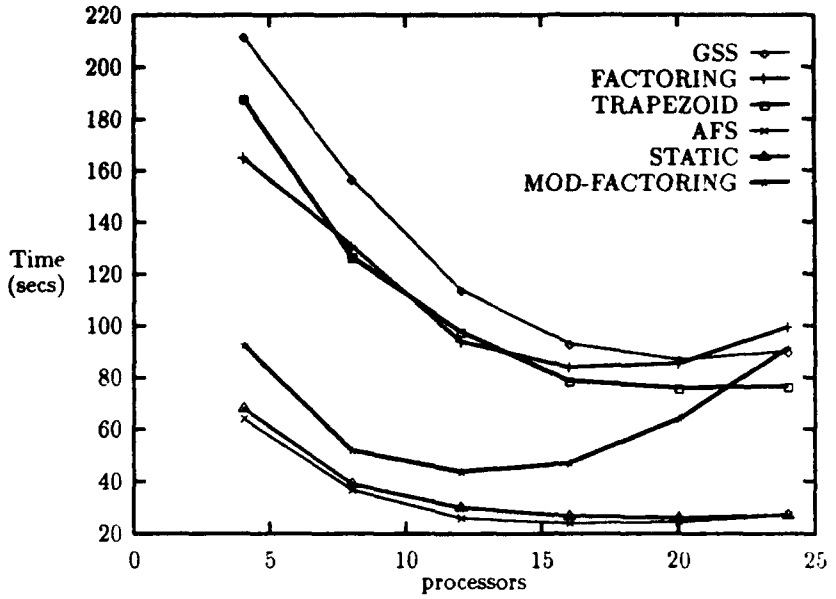

Figure 5.15: Gaussian elimination on the KSR-1.

short-term fluctuations cause some processors to execute iterations that belong to other processors, and almost all affinity is lost.

Figure 5.16 shows the completion time of the transitive closure application under different scheduling algorithms. We note that all previously known schedulers saturate at 10-15 processors and their performance gets worse beyond this point. The communication and synchronization overhead that these schedulers impose at 10-15 processors is so high, that adding more processors just makes matters worse. Once again, AFS performs the best of all algorithms. The closest algorithm to AFS is TRAPEZOID, which has a small number of synchronization operations and manages to degrade performance more gracefully. Although AFS performs better than the other algorithms, the gap between AFS and the other algorithms is not as great as it was for Gaussian elimination. There is a simple explanation for this: transitive closure has significant load imbalance, while Gaussian elimination does not. In order to balance the load, AFS must migrate (reassign) iterations from one processor to another. This migration results in a loss of affinity. In Gaussian elimination, AFS almost never has to migrate iterations because there is almost no imbalance among the processors, and thus no loss of affinity.

Figure 5.17 presents the completion time of SOR on the KSR-1. Once again, AFS, STATIC, and MOD-FACTORING are the best algorithms. What is unusual is that they are not much better than the others. SOR has *lots* of affinity to be preserved, and there

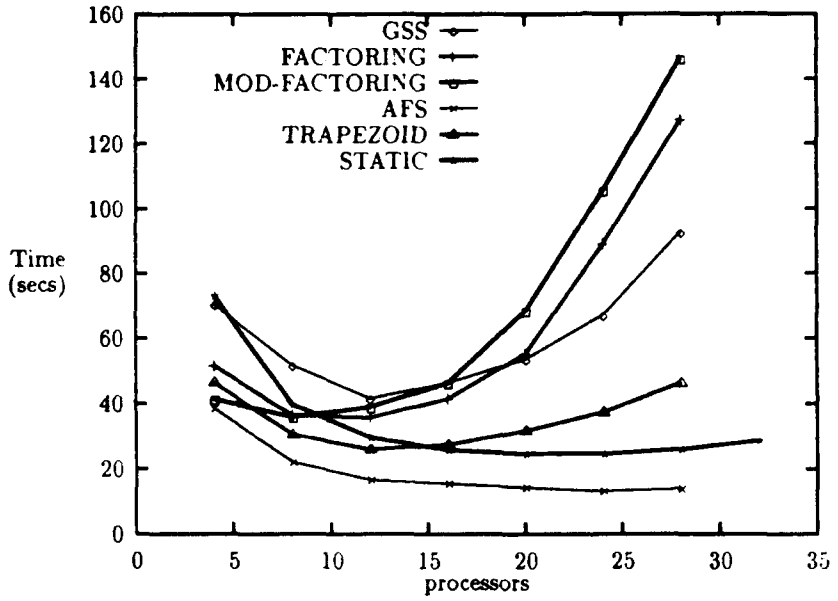

Figure 5.16: Transitive closure on the KSR-1. (1024 node graph, where 40% of them form a clique).

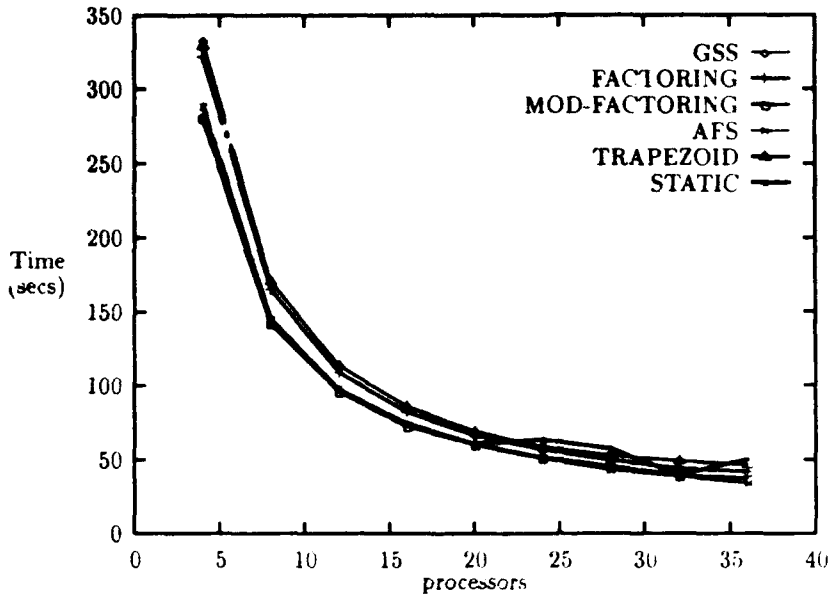

Figure 5.17: SOR on the KSR-1. (1024 by 1024, 128 iterations)

is almost no imbalance to hinder the preservation of affinity. So, why isn't AFS much better than the other schedulers? The reason lies with a fundamental computation cost that changed in the KSR-1. SOR performs a few additions and one division inside the inner loop. Although floating point addition is supported in hardware on the KSR-1, floating point division is implemented in software. Thus, the KSR-1 is a relatively slow machine for applications that make heavy use of floating point division (like SOR). Any benefits that arise from preserving affinity are bound to be a small percentage of the total completion time of this application on this machine.

Comparing Affinity Preserving Algorithms

Now that we have demonstrated the importance of affinity, we compare the three affinity-preserving loop schedulers we considered: static, affinity scheduling and modified-factoring. Static scheduling preserves affinity but fails to balance the load in the cases where load imbalance arises. Thus, it is inappropriate in dynamic environments. In several cases modified-factoring preserves almost all the affinity there is, but in some cases it is highly unstable. This behavior can be traced to two properties of this algorithm:

- Modified-factoring has high synchronization overhead, much higher than factoring, because each processor searches for a chunk of *preferred* work in a central queue

where all chunks are held. In contrast, most other algorithms, including factoring, take the first chunk they find, and thus have smaller synchronization overhead.

- Modified-factoring is *too* eager to balance the load. If a processor does not find a preferred chunk, it takes the preferred chunk of another processor, which in turn takes the chunk of another processor and so on. Thus, if there is even one processor that finishes its phase before the others start, it will probably result in no chunk being assigned to its preferred processor.

Affinity scheduling overcomes the problems of the other affinity preserving algorithms, by being a dynamic scheduling algorithm that reassigns iterations only when there are no more local iterations to execute. This makes affinity scheduling both flexible and robust at the same time.

5.4.9 Large Problems

Affinity scheduling performs the best when the working set of the parallel application is in the caches (or local memories) of the multiprocessor. Although both caches and local memories used to be rather small, and could not hold the input for large scientific problems, this is no longer the case. Local memories have become larger every year, quadrupling their size every three years [37]. Even caches continue to grow in size: the SGI Iris has 1 MB caches, while the KSR-1 has 32 MB caches.

To verify that affinity scheduling results in significant performance improvements even for very large problems, we ran Gaussian elimination on a 4096×4096 matrix on 16 processors on the KSR-1. This problem needs more than five hours to complete on one KSR-1 processor, and about 25 days to complete on one Butterfly I processor. We ran the application on the KSR-1 and its completion time under various loop schedulers was:

loop scheduler	completion time (minutes) 16 KSR-1 processors
AFS	20.6
STATIC	20.9
MODIFIED-FACTORING	22.7
FACTORING	47.3
TRAPEZOID	50.7
GSS	73.7

We see that even for very large scientific problems, affinity scheduling results in significant performance improvements, a factor of 2.5 over FACTORING and TRAPEZOID, and a factor of 3.5 over GSS. All affinity preserving schedulers (STATIC, AFS and MODIFIED-FACTORING) perform within 10% of each other.

5.4.10 Summary of Results

Our experimental results demonstrate that the affinity scheduling algorithm has load balancing properties comparable to those of the best known loop scheduling algorithms (i.e., guided self-scheduling, trapezoid, factoring), while maintaining processor affinity, and thereby significantly reducing communication overhead. The number of synchronization operations per queue required by affinity scheduling is not much larger than the number of operations required by the trapezoid algorithm, which induces the least amount of synchronization overhead of the dynamic algorithms. Moreover, the number of *serialized* synchronization operations induced by affinity scheduling is always less than the number of *serialized* synchronization operations required by the other dynamic methods. As a result, in most cases affinity scheduling performs better than any other known algorithm on modern shared-memory multiprocessors.

5.5 Related Issues

Loop scheduling can be viewed as part of the general problem of scheduling tasks in multiprocessor systems so as to minimize the completion time of parallel applications. In this context, loop scheduling is analogous to process scheduling, which is concerned with many of the same issues, including load imbalance, synchronization overhead, and communication overhead.

While some of the results regarding process scheduling apply to loop scheduling, there is an important distinction between the two problem domains: a loop scheduling algorithm must choose an appropriate decomposition (i.e., chunks), while the process scheduling algorithm is given the decomposition selected by the programmer as input. Thus, loop scheduling considers an additional dimension — loop decomposition so as to minimize load imbalance — and therefore must make tradeoffs in three dimensions (synchronization overhead, communication overhead, load imbalance). It is this third dimension that distinguishes work in loop scheduling, and that has dominated loop scheduling research. However, even when there is a straightforward decomposition, and the loop scheduling problem degrades to the process scheduling problem, process scheduling algorithms are of little use, since communication is usually ignored in process scheduling. In those cases where the role of communication has been considered, the resulting schedulers need large amounts of information, like call graphs, detailed communication patterns, and exact communication costs [14, 58]. Such knowledge is not generally available, as it may depend on unpredictable run-time factors (e.g. the input of the application). Dynamic scheduling algorithms, including affinity scheduling, emphasize algorithms and heuristics that are robust in dynamic environments and work under limited information.

Another aspect of loop scheduling that has attracted some attention deals with scheduling loops that have dependencies within the statements of one iteration, across iterations, or both [47, 58]. This problem is interesting and leads to challenging graph-theoretic problems. Unfortunately the most general form of the problem is intractable and must be solved using heuristics. Moreover, applications that have loops with depen-

dencies do not usually achieve large speedups as they can execute in parallel at most a few iterations at a time. Unless each iteration takes a long time to execute, and can be easily parallelized, loops with dependencies are of limited use in significantly enhancing application performance using parallel processing.

Recent compiler work has also dealt with the issue of communication through the use of blocking or tiling [77]. Blocking is a method of restructuring a loop so that successive accesses to data by a processor occur as close in time as possible, so that the data is still in the local cache when it is referenced over and over. Blocking can be used at any level of the memory hierarchy, but it has the potential to introduce load imbalance, since large blocks imply not only improved cache hit ratios, but also coarse computation granularity, which may result in load imbalance.

5.6 Conclusions

In this chapter we discussed several important properties of loop scheduling algorithms. Most algorithms assign the iterations of a loop to processors so as to minimize load imbalance, while incurring minimum synchronization overhead. We argued that the non-uniform access time to data in a shared-memory multiprocessor (due to local caches or memory) introduces a new dimension to the loop scheduling problem: communication overhead. We showed that traditional loop scheduling methods that ignore communication costs impose significant overhead on parallel applications, and proposed a new loop scheduling algorithm based on affinity scheduling. This new algorithm performed better than all other known algorithms in our experiments. The main ideas underlying affinity scheduling are:

- Affinity scheduling uses per-processor work queues, instead of a central work queue. Accesses to several work queues may proceed in parallel, and most accesses to work queues are local access that do not suffer from contention. Synchronization across processor occurs only if load imbalance arises.
- When a parallel loop is embedded within a sequential loop (a common case), affinity scheduling assigns an iteration of the loop to the same processor each time it is executed. If the iteration accesses the same data each time, then the data will already be in the local memory or cache, reducing communication overhead.

Based on our experiments with affinity scheduling and other loop scheduling policies on three different multiprocessors, we conclude:

- ***Loop scheduling algorithms should simultaneously consider load imbalance, synchronization, and communication.*** Known policies that ignore communication overhead incur a significant performance penalty in current multiprocessors. If processor speeds continue to improve more quickly than memory or interconnection speeds, communication will be an increasing percentage of an application's execution time; scheduling methods that reduce both communication and synchronization overhead are going to have an even greater impact in the future.

- *Affinity scheduling is robust.* Our experiments cover a range of applications with widely varying characteristics. For applications that create affinity between iterations and processors, affinity scheduling is by far the best algorithm. For applications with a lot of input-dependent load imbalance (i.e., transitive closure), affinity scheduling is again the best scheduling algorithm. Even for applications that have no affinity to exploit, but exhibit significant potential for load imbalance (i.e., adjoint convolution and L4), affinity scheduling is among the best algorithms.

- *Central work queues are a bottleneck, even in small-scale multiprocessors.* Central work queues (or ready queues) have been criticized for serializing access to work, which can produce a bottleneck in large-scale systems. Efficient synchronization primitives (e.g., fetch and ϕ) and efficient chunking algorithms can help with synchronization overhead, but the problem of communication overhead remains. Central work queues require the frequent movement of data among processors, since every process must load the data it needs into the local cache. The resulting communication overhead degrades performance even for a very small number of processors.

In summary, our theoretical and experimental evaluation shows that affinity scheduling has the attractive load balancing properties of the best known loop scheduling algorithms, but also reduces communication overhead substantially. This overhead is quite high on current multiprocessors, and is likely to increase in the future. We conclude that loop scheduling techniques, such as affinity scheduling, that minimize communication overhead will be increasingly important in the future.

6 Conclusions and Future Work

In this dissertation we studied the problems of scheduling and decomposition for locality in shared-memory multiprocessors. We used several applications and several machines, including the most popular models of Encore, Sequent, SGI, and Butterfly multiprocessors. Based on our experimental and analytical evaluation we conclude:

- *Communication is a significant and increasing source of overhead in shared-memory multiprocessors.* Communication was not an issue in shared-memory multiprocessors 10 or even 5 years ago, mainly because processors were so slow that communication was a negligible percentage of the total completion time of the parallel application. An impressive increase in processor speeds accompanied by a moderate increase in memory and interconnection network speeds has resulted in an increase in the (relative) cost of communication by more than an order of magnitude. Our experiments with several multiprocessors suggest that the oldest (and slowest) multiprocessor (the BBN Butterfly I) has the best performance (in terms of speedup and efficiency), while the most recent multiprocessors (the Butterfly TC2000 and the SGI Iris) have the worst performance, having efficiency as low as 30% in many cases. The low efficiency is particularly interesting in light of the fact that we used small scale multiprocessors (up to 50 processors). Had we used larger scale multiprocessors we would have measured even greater performance degradation.

- *Scheduling and decomposition methods that reduce communication are essential.* Previously known scheduling and decomposition algorithms (especially dynamic algorithms) have, for the most part, ignored communication as an overhead dimension. As a result, programming techniques that were efficient on the previous generation of shared-memory multiprocessors now impose unacceptable overhead. For example, even though thread libraries have dramatically reduced the cost of thread creation [5, 12, 71], lightweight threads increase communication, since upon creation, each thread must load its working set into the local cache or memory. This additional communication can easily increase the overhead associated with each thread by two orders of magnitude, making the use of lightweight threads very inefficient on modern shared-memory machines. Thus, the solid body of research that deals with lightweight threads, including operating system support for threads, performance analysis of multi-threaded applications, and compiler

implementations that use multiple threads, is of limited applicability. Without scheduling algorithms that reduce communication, multi-threaded programming models are impractical on modern shared-memory multiprocessors.

- *Our proposed scheduling algorithms (memory-conscious scheduling and affinity scheduling) show significant performance improvements now, and will those improvements are expected to increase in the future.* We implemented our thread scheduling method (memory-conscious scheduling) and compiler loop scheduling method (affinity scheduling) on several multiprocessors, including the Butterfly family of parallel processors, and the SGI Iris and Sequent Symmetry bus-based cache-coherent multiprocessors. We compared our algorithms against the best known algorithms, and showed that our algorithms have significant performance improvements over other popular methods, especially on the recent generation of multiprocessors. Other algorithms do not address communication costs, which have only recently become the dominant source of overhead. In addition, many previous studies of scheduling algorithms (both experimental and simulation based) used skeletons of applications rather than real applications [5, 47, 26, 59, 73]. Although these skeletons accurately represent the structure of applications, they don't make memory references, which tends to obscure the cost of communication in these studies.

Given that current and future multiprocessors will have a deep memory hierarchy, system software must take this hierarchy into account. There are several extensions to our work that address this problem:

- All of our work has dealt with a two-level memory hierarchy (local and non-local memory). This work could be extended to multi-level memory hierarchies. In such architectures, there exists local, close, and distant memory for each processor. A multi-level memory hierarchy presents a rich variety of choices for scheduling and data allocation, but also complicates these two problems.

- Parallelizing compilers have traditionally focused on finding the maximum parallelism available in a program, while generally ignoring the memory hierarchy. Although recent work has begun to deal with the memory hierarchy explicitly (e.g., [77]), there is still much to be done. Several decisions made by parallelizing compilers have to be reconsidered and possibly redesigned. Scheduling, decomposition, data placement, process placement, and loop optimization have to be carefully judged in terms of their effect on moving data across the memory hierarchy. Compilers may also be able to exploit information about data reference patterns gathered from previous runs of the same application.

- It is becoming increasingly apparent that parallel software is fundamentally different from sequential software, and yet parallel processors are built out of the same components as uniprocessors. New hardware techniques may be needed for multiprocessors. Latency-tolerant processors [2] are one step in this direction, but other ideas need to explored.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development". In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pages 93–112, Pittsburgh, PA, June 1986.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. "APRIL: A Processor Architecture for Multiprocessing". In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] S. Ahuja, N. Carriero, and D. Gelernter. "Linda and Friends". *IEEE Computer*, 19(8):26–34, August 1986.

[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 53–79, October 1991.

[5] T. E. Anderson, E. D. Lazowska, and H. M. Levy. "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors". *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[6] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. "The Interaction of Architecture and Operating System Design". In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[7] J. Archibald and J.-L. Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[8] T. S. Axelrod. "Effects of Synchronization Barriers on Multiprocessor Performance". *Parallel Computing*, 3:129–140, 1986.

[9] BBN Advanced Computers Inc. *The Uniform System Approach To Programming the Butterfly Parallel Processor.* Cambridge, Massachusetts, 1986. BBN Report 6149, Version 2.

[10] BBN Advanced Computers Inc. *Chrysalis Programmers Manual.* Cambridge, MA, February 1988.

[11] B.N. Bershad, E.D. Lazowska, and H.M. Levy. "PRESTO: A System for Object-Oriented Parallel Programming". *Software - Practice and Experience*, 18(8):713–732, August 1988.

[12] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D.B. Wagner. "An Open Environment for Building Parallel Programming Systems". In *Proceedings of the ACM/SIGPLAN PPEALS 1988 Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 1–9, July 1988.

[13] D. L. Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer*, 23(5):35–43, May 1990.

[14] S. H. Bokhari. *Assignment problems in parallel and distributed computing.* Kluwer Academic Publishers, Boston, 1987.

[15] W. J. Bolosky and M. L. Scott. "A Trace-Based Comparison of Shared Memory Multiprocessor Architectures". Technical Report 432, University of Rochester, Computer Science Department, July 1992.

[16] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. "NUMA Policies and Their Relation to Memory Architecture". In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[17] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. "Simple But Effective Techniques for NUMA Memory Management". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[18] E. D. Brooks. "The Butterfly Barrier". *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[19] D. R. Cheriton, H. A. Goosen, and P. Machanick. "Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience". In *Proceedings of the International Symposium on Shared-Memory Multiprocessing*, pages 109–118, 1991.

[20] A.L. Cox and R.J. Fowler. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[21] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. "Multiprogramming on Multiprocessors". In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, Dallas, Texas, December 1991.

[22] S. Dandamudi. "A Comparison of Task Scheduling Strategies for Multiprocessor Systems". In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 423–426, Dallas, Texas, December 1991.

[23] T. W. Doeppner Jr. "Threads: A System for the Support of Concurrent Programming". Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[24] T. H. Dunigan. "Kendall Square Multiprocessor: Early Experiences and Performance". Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, May 1992.

[25] D. L. Eager, E.D. Lazowska, and J. Zahorjan. "The Limited Performance Benefits of Migrating Active Processes for Load Sharing". In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 63–72, May 1988.

[26] D. L. Eager and J. Zahorjan. "Adaptive Guided Self-Scheduling". Technical Report 92-01-01, Department of Computer Science and Engineering, University of Washington, January 1992.

[27] D.L. Eager and J. Zahorjan. "Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing". *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[28] J. Edler, J. Lipkis, and E. Schonberg. "Process Management for Highly Parallel UNIX Systems". Technical Report Ultracomputer Note 136, Ultracomputer Research Laboratory, New York University, April 1988.

[29] D.G. Feitelson and L. Rudolph. "Distributed Hierarchical Control for Parallel Processing". *IEEE Computer*, 23(5):65–77, May 1990.

[30] R.J. Fowler and L. Kontothanasis. "Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing". Technical Report 411, University of Rochester, Computer Science Department, 1992.

[31] R. Goldman and R. P. Gabriel. "Qlisp: Parallel Processing in Lisp". *IEEE Software*, 6(4):51–59, July 1989.

[32] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. "Comparative Evaluation of Latency Reducing and Tolerating Techniques". In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, May 1991.

[33] A. Gupta, A. Tucker, and S. Urushibara. "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications". In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, May 1991.

[34] R. Gupta. "Synchronization and Communication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systems". In *Proceedings of the International Conference on Parallel Processing*, pages II:23–30, August 1989.

[35] R. Gupta and C. R. Hill. "A Scalable Implementation of Barrier Synchronization Using An Adaptive Combining Tree". *International Journal on Parallel Programming*, 18(3):161–180, June 1989.

[36] R. H. Halstead Jr. "Multilisp: A Language for Concurrent Symbolic Computation". *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[37] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[38] D. Hensgen, R. Finkel, and U. Manber. "Two Algorithms for Barrier Synchronization". *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[39] M. A. Holliday. "Reference History, Page Size, and Migration Daemons in Local/Remote Architectures". In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989.

[40] S.F. Hummel, E. Schonberg, and L.E. Flynn. "Factoring: A Practical and Robust Method for Scheduling Parallel Loops". *Communications of the ACM*, 35(8):90–101, August 1992.

[41] C.P. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors". *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[42] R. P. LaRowe Jr. and C. S. Ellis. "OS Experimentation and a User Community Coexist Under the DUnX Kernel". In *Proceedings of the International Conference on Parallel Processing*, pages II–158–II–166, August 1991.

[43] R. P. LaRowe, Jr. and C. S. Ellis. "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors". *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[44] R. P. LaRowe Jr., J. T. Wilkes, and C. S. Ellis. "Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor". *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 122–132, April 1991.

[45] S. T. Leutenegger. *Issues in Multiprogrammed Multiprocessor Scheduling.* PhD thesis, University of Wisconsin-Madison, August 1990.

[46] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. "Policy/Mechanism Separation in Hydra". In *Proceedings of the 5th Symposium on Operating Systems Principles*, pages 132–140, Austin, TX, November 1975.

[47] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[48] S.-P. Lo and V.D. Gligor. "Properties of Multiprocessor Scheduling Algorithms". In *Proceedings of the International Conference on Parallel Processing*, August 1987.

[49] S.-P. Lo and V.D. Gligor. "A Comparative Analysis of Multiprocessor Scheduling Algorithms". In *Proceedings 7th International Conference on Distributed Computing Systems*, pages 205–222, September 1987.

[50] B. Lubachevsky. "Synchronization Barrier and Related Tools for Shared Memory Parallel Programming". In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II:175–179, August 1989.

[51] S. Lucco. "A Dynamic Scheduling Method for Irregular Parallel Programs". In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 200–211, June 1992.

[52] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. "The Effects of Multiprogramming on Barrier Synchronization". In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

[53] B.D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P. Markatos. "First Class User-Level Threads". In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[54] C. McCann, R. Vaswani, and J. Zahorjan. "A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors". Technical Report 90-03-02, Department of Computer Science and Engineering, University of Washington, March 1990 (Revised February 1991).

[55] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[56] J. K. Ousterhout. "Scheduling Techniques for Concurrent Systems". In *Proceedings of Distributed Computing Systems*, pages 22–30, October 1982.

[57] John Ousterhout. "Why Aren't Operating Systems Getting Faster as Fast as Hardware?". *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[58] C. D. Polychronopolous. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Boston, MA, 1988.

[59] C. D. Polychronopoulos and D. J. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers". *IEEE Transactions on Computers*, C-36(12), December 1987.

[60] E. Rothberg and A. Gupta. "Parallel ICCG on a Hierarchical Memory Multiprocessor - Addressing the Triangular Solve Bottleneck". Technical Report CSL-TR-90-449, Stanford University, September 1990.

[61] V. Sarkar. *Partitioning and Scheduling for Execution on Multiprocessors.* PhD thesis, Stanford University, April 1987.

[62] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors". Technical Report 309, University of Rochester, Computer Science Department, March 1989.

[63] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "Multi-Model Parallel Programming in Psyche". In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[64] J.P. Singh, W-D. Weber, and A. Gupta. "SPLASH: Stanford Parallel Applications for Shared-Memory". *Computer Architecture News*, 20(1):5–44, March 1992.

[65] B. Smith. "Architecture and Applications of the HEP Computer System". In *Proceedings of the SPIE, Real-Time Signal Processing IV*, 1981.

[66] M. S. Squillante. *Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation.* PhD thesis, Department of Computer Science and Engineering, University of Washington, October 1990.

[67] M. S. Squillante and E.D. Lazowska. "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling". Technical Report 89-06-01, Computer Science Department, University of Washington, February 1990.

[68] M. S. Squillante and R. D. Nelson. "Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling". In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 143–155, May 1991.

[69] Sun Microsystems, Inc. "Lightweight Processes". In *SunOS Programming Utilities and Libraries*, March 1990. Sun Part Number 800-3847-10.

[70] P. Tang and P.-C. Yew. "Processor Self-Scheduling for Multiple Nested Parallel Loops". In *Proceedings 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[71] R.H. Thomas and W. Crowther. "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors". In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[72] A. Tucker and A. Gupta. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[73] T.H. Tzen and L.M. Ni. "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers". *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[74] R. Vaswani and J. Zahorjan. "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors". In *Proceedings of the Thirteenth Sumposium on Operating Systems Principles*, pages 26–40, October 1991.

[75] W.-D. Weber and Anoop Gupta. "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results". In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 273–280, Jerusalem, Israel, May 1989.

[76] M. Weiser, A. Demers, and C. Hauser. "The Portable Common Runtime Approach to Interoperability". In *Proceedings of the 12th Sumposium on Operating Systems Principles*, pages 114–122, December 1989.

[77] M. E. Wolf and M. S. Lam. "A Data Locality Optimizing Algorithm". In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[78] P.-C. Yew, N.-F. Tzeng, and D.H. Lawrie. "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors". *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[79] J. Zahorjan, E. D. Lazowska, and D. L. Eager. "Spinning Versus Blocking in Parallel Systems with Uncertainty". Technical Report 88-03-01, Department of Computer Science and Engineering, University of Washington, March 1988.

[80] J. Zahorjan, E.D. Lazowska, and D.L. Eager. "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems". *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

[81] J. Zahorjan and C. McCann. "Processor Scheduling in Shared Memory Multiprocessors". In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.